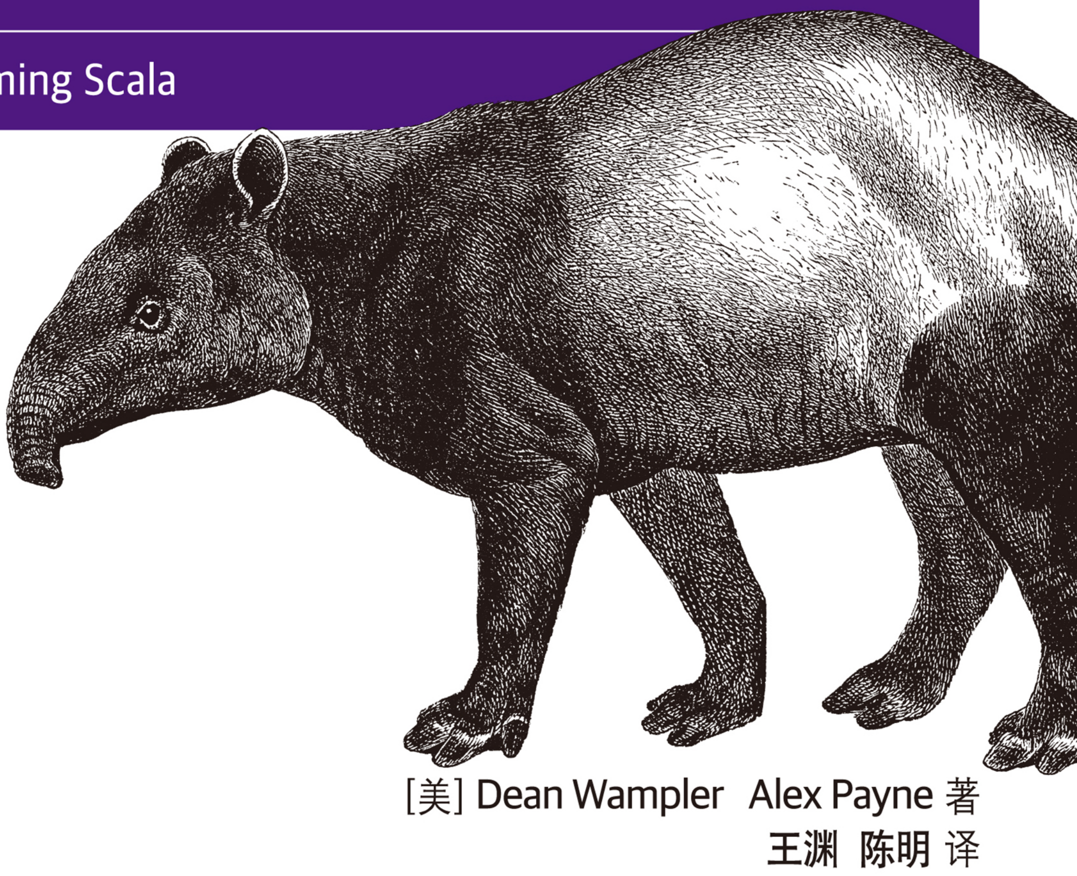
O'REILLY®

TURING 图灵程序设计丛书

第2版

# Scala 程序设计

Programming Scala



[美] Dean Wampler　Alex Payne 著
王渊　陈明 译

中国工信出版集团　人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书

# Scala程序设计（第2版）

## Programming Scala

[美] Dean Wampler Alex Payne 著
王渊 陈明 译

O'REILLY®
*Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo*
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（C I P）数据

Scala程序设计 : 第2版 / (美) 万普勒
(Wampler,D.）, (美) 佩恩 (Payne,A.）著 ; 王渊, 陈
明译. -- 北京 : 人民邮电出版社, 2016.3
（图灵程序设计丛书）
ISBN 978-7-115-41681-0

Ⅰ. ①S… Ⅱ. ①万… ②佩… ③王… ④陈… Ⅲ. ①
JAVA语言－程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2016)第023309号

## 内 容 提 要

本书通过大量的代码示例，全面介绍Scala这门针对JVM的编程语言，向读者展示了如何高效地利用Scala语言及其生态系统，同时解释了为何Scala是开发高扩展性、以数据为中心的应用程序的理想语言。

本书既适合Scala初学者入门，也适合经验丰富的Scala开发者参考。

◆ 著 [美] Dean Wampler Alex Payne
译 王 渊 陈 明
责任编辑 岳新欣
执行编辑 刘 敏
责任印制 杨林杰

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号
邮编 100164 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京 印刷

◆ 开本：800×1000 1/16
印张：31.25
字数：761千字 2016年3月第1版
印数：1－3 000册 2016年3月北京第1次印刷

著作权合同登记号 图字：01-2015-6359号

定价：109.00元

**读者服务热线：(010)51095186转600 印装质量热线：(010)81055316**

**反盗版热线：(010)81055315**

**广告经营许可证：京崇工商广字第0021号**

# O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

## 业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

# 目录

# 序

作为一名程序员，我的职业生涯中一直贯穿着这样的主题：寻求更好的抽象和更好的工具来编写更好的软件。经过了这些年，我认为可组合性（composability）是一项比其他特征更重要的特征。如果我们编写的代码具有很好的可组合性，这通常意味着这些代码同样具备软件工程师所看重的其他特征，如正交性（orthogonality）、松耦合性以及高聚合性（high cohesion）。这些都是互通的。

几年前，当我发现 Scala 语言时，它的可组合性便给我带来了很大的震撼。

Martin Odersky 创造 Scala 时，运用了一些简洁的设计方法以及源于面向对象和函数式编程的一些看似简单却很强大的抽象，这使得 Scala 具备高聚合性，而具备了正交性的高度抽象则给这门语言带来可用于软件设计各个方面的可组合性。Scala 是一门真正具备了可扩展性的语言，我们既能使用它编写各种脚本语言，也能使用它实现大规模企业应用和中间件。

Scala 起源于学术界，却已经成长为了一门注重实用性的语言，对于那些真实生产环境中的应用场景，Scala 已经完全准备好了。

《Scala 程序设计》一书的实用性让我感到兴奋。Dean 干得太棒了，除了使用有趣的讨论和示例对 Scala 这门语言进行讲解之外，还将这些内容套到真实世界的应用场景中。这本书是为那些希望能够解决实际问题的程序员所编写的。

几年前，我们还都是面向切面编程委员会的成员时，我认识了 Dean。我很庆幸能够认识他。Dean 拥有一个少见的混合型大脑，他既能思考高深的学术问题，也能想到如何运用实际的方法解决问题。

通过阅读这本书，你将学到如何使用 mixin 和函数组合编写可重用组件；如何运用 Akka 库编写响应式（reactive）应用；如何高效地使用 Scala 提供的一些高级特征，如宏、higher kinded 类型；如何通过 Scala 的丰富、灵活而又富有表现力的语法构造领域特定语言；如何有效地测试你的 Scala 代码；如何通过 Scala 简化大数据问题，等等。

读者们，请好好享受阅读这本书的时光，正如我所做的那样。

——Jonas Bonér

Typesafe 公司联合创始人兼技术总监，2014 年 8 月

# 前言

《Scala 程序设计》向读者介绍了一门既令人振奋又功能强大的语言，该门语言集合了现代对象模型、函数式编程以及先进类型系统的所有优点，同时又能应用获得产业界大量投资的 Java 虚拟机（JVM）。这本书通过大量的代码示例，向读者全面阐述了如何使用 Scala 迅速编写代码，解释了为什么 Scala 是编写可扩展、分布式、基于组件且支持并发和分布的应用程序的最完美语言。Scala 运行在先进的 JVM 平台之上，通过阅读本书，读者还能了解到 Scala 是如何发挥 JVM 平台优势的。

如果你想了解更多内容，请访问 http://programming-scala.org 或查阅本书目录（http://shop.oreilly.com/product/0636920033073.do）。

## 欢迎阅读《Scala程序设计（第2版）》

本书第 1 版出版于 2009 年秋，是当时市面上第三本讲述 Scala 的图书，仅仅因为耽误了几个月，未能成为第二本。Scala 当时的官方版本号为 2.7.5，而 2.8.0 版则接近完工。

从那时起，Scala 世界发生了巨大的变化。编写本书时，Scala 的版本号为 2.11.2。为了进一步提升 Scala 语言以及相关工具，Scala 的创建者 Martin Odersky 与基于 actor 模型的并发框架 Akka 的作者 Jonas Bonér 一同创立了 Typesafe（http://typesafe.com）公司。

现在已经出版了很多 Scala 的图书，我们真的有必要再推出第 2 版吗？市场上现存不少适合初学者的 Scala 指南，也出现了一些供高级学习者使用的图书，由 Artima 出版 Odersky 等人撰写的《Scala 编程（第 2 版）》仍被视为 Scala 语言的百科全书。

然而，本书第 2 版非常完整地描述了 Scala 语言及其生态系统，既为初学者成为 Scala 高级用户提供了所需要的指导，又关注了开发人员所面对的实用性问题，这是本书独一无二的地方，也是第 1 版广受欢迎的原因所在。

与 2009 年相比，现在有更多的机构选用了 Scala，大多数 Java 开发者也都听说过这门语言。同时，也出现了一些针对 Scala 语言的持续质疑。Scala 是不是太复杂了？ 既然 Java 8 已经引入了一些 Scala 特性，那还有必要使用 Scala 吗？

对于真实世界中的种种质疑，我将一一解答。我常常说，Scala 的一切，包括它的不足之处，都让我为之着迷。我希望读者阅读完本书也会有同感。

## 如何阅读本书

本书论述全面，初级读者无须阅读所有内容便可以使用 Scala 进行编程。本书的前 3 章“零到六十：Scala 简介”“更简洁，更强大”和“要点详解”，简要概括了 Scala 的核心语言特性。第 4 章“模式匹配”和第 5 章“隐式详解”描述了使用 Scala 编程时每天都会用到的两类基本工具，通过对这两类工具的描述将读者引领到更深的领域里。

函数式编程（FP）是一种重要的软件开发方案，假如你之前从未接触过 FP，那么阅读第 6 章能通过 Scala 学习函数式编程。紧接着第 7 章，将说明 Scala 对 `for` 循环的扩展，以及如何使用该扩展提供的简洁语法实现复杂而又符合规范的函数式代码。

之后，第 8 章将介绍 Scala 是如何支持面向对象编程（OOP）的。为了强调 FP 对于解决软件开发问题的重要性，我将 FP 相关章节放到 OOP 章节之前。如果将 Scala 当作“更好的面向对象的 Java”，那会较容易上手，但这样会丢掉这门语言最有力的工具。第 8 章的大多数内容在概念上很容易理解，读者将学到在 Scala 中如何定义类、构造函数等与 Java 相似的概念。

第 9 章将继续探索 Scala 的功能——使用 trait 对行为进行封装。Java 8 受到了 Scala trait 机制的影响，通过对接口进行扩展，新增了部分 trait 功能。对于这部分内容而言，即便是有经验的 Java 程序员也需要花时间理解。

接下来的 4 章，从第 10 章到第 13 章，“Scala 对象系统（I）”“Scala 对象系统（II）”“Scala 集合库”以及“可见性规则”，详细地讲解了 Scala 的对象模型和库类型。由于第 10 章包含了一些必须要尽早掌握的基本知识，因此阅读时要务必仔细。第 11 章讲述了如何正确地实现普通类型层次，你可以在第一遍阅读本书时略过这一章。第 12 章讨论了集合设计问题并提供了合理使用集合的相关信息。再重申一遍，假如你初次接触 Scala，那么请先略过此章，当你试图掌握集合类 API 的详细内容时，再回来学习。最后，第 13 章详细解释了 Scala 是如何对 Java 的 public、protected 以及 private 可见性概念进行细粒度扩展的。可以快速阅览此章。

从第 14 章开始，我们将进入更高级的主题：Scala 复杂类型。这部分内容划分为两章：第 14 章包含了 Scala 新手相对容易理解的概念，而第 15 章则讲述了更高级的内容，你可以选择以后再进行阅读。

类似地，第 16 章“高级函数式编程”讲述的内容中包括了更多高级的理论概念，例如，Monad 和仿函式（Functor）这些起源于范畴论的概念。一般水平的 Scala 开发者在最初并不需要掌握这些内容。

第 17 章“并发工具”有助于开发大型服务的程序员实现并发性的可伸缩性和可扩展性。这一章既论述了 Akka 这一基于 actor 的富并发模型，又讲述了像 `Future` 这类有助于编写异步代码的库类型。

第 18 章“Scala 与大数据”，通常而言，在大数据以及其他以数据为中心的计算领域里，应用 Scala 和函数式编程能够构造杀手级应用。

第 19 章“Scala 动态调用”和第 20 章“Scala 的领域特定语言”是较为高级的专题，探讨了可用于构建富领域特定语言的一些工具。

第 21 章“Scala 工具和库”讨论了一些 IDE 和第三方库。假如你是 Scala 新手，那么请阅读 IDE 和编辑器支持的相关小节，同时阅读关于 Scala 公认的项目构建工具：SBT 的相关小节。本章最后列出了可以引用的库列表。第 22 章“与 Java 的互操作”对于那些需要互用 Java 和 Scala 代码的团队而言很有帮助。

第 23 章“应用程序设计”是为架构师和软件组长而写的。我在这一章分享了自己在应用设计方面的一些观点。传统模式使用了相对较大的 JAR 文件，而这些 JAR 文件又包含了复杂的对象图谱。因此我认为这种模式是一种不良模式，需要进行变更。

最后，第 24 章“元编程：宏与反射”介绍了本书最高级的主题。当然，如果你是初学者，也可以略过这一章。

本书在附录 A 中总结了一些资料，供读者进一步阅读。

## 本书未涉及的内容

模块化库是 Scala 最新的 2.11 版的一大焦点，它将库文件分解成更小的 JAR 文件，这样一来，在将系统部署到空间受限的环境时（如手机设备），便能很容易移除不需要的代码。除此之外，新版移除了库中一些原本被标示为“过时”（deprecated）的包和类型，还将其他的一些包和类型标示为 deprecated，这通常是因为 Scala 不再维护这些包和类型，而且有更好的第三方的替代品。

因此，我们不会在本书中讨论那些在 2.11 版本中被标示为 deprecated 的包，具体如下。

- `scala.actors`（http://www.scala-lang.org/api/current/scala-actors/#scala.actors.package）
  一套 actor 库。请使用 Akka actor 库。（我们将在 17.3 节对该库进行描述。）

- `scala.collection.script`（http://www.scala-lang.org/api/current/#scala.collection.script.package）
  该库用于编写监控集合以及更新集合相关“脚本”。

- `scala.text`（http://www.scala-lang.org/api/current/#scala.text.package）
  一套用于“格式化打印”（pretty-printing）的库。

下面列举了在 Scala 2.10 中标示为 deprecated 并已从 2.11 版移除的包。

- `scala.util.automata`（http://www.scala-lang.org/api/2.10.4/#scala.util.automata.package）
  使用正则表达式构建确定有限自动机（DFA）。

- `scala.util.grammar`（http://www.scala-lang.org/api/2.10.4/#scala.util.grammar.package）
  属于 parsing 库。

- `scala.util.logging`（http://www.scala-lang.org/api/2.10.4/#scala.util.logging.package）
  推荐使用某一 JVM 平台上活跃的第三方日志库。

- `scala.util.regexp`（http://www.scala-lang.org/api/2.10.4/#scala.util.regexp.package）
  对正则表达式进行句式分析。`scala.util.matching` 包同样支持正则表达式，请使用功能更为强大的 `scala.util.matching` 包。

- .NET 编译器后台
  Scala 团队曾在 .NET 运行的环境之上搭建编译器后台及库。不过由于大家对这次迁移的兴趣不断衰减，因此这项工作已经暂停。

我们不会对 Scala 库中每个包和类型都进行讨论。由于篇幅和其他原因，下面这些包并不会在本书中提及。

- `scala.swing`（http://www.scala-lang.org/api/current/scala-swing/#scala.swing.package）
  对 Java Swing 库进行封装。尽管仍然有人维护该库，但已很少有人使用它。

- `scala.util.continuations`（http://www.scala-lang.org/files/archive/api/current/scala-continuations-library/#scala.util.continuations.package）
  编译器插件，用于生成连续传递格式（continuation-passing style，CPS）的代码。这是一个特殊的工具，目前很少有人使用它。

- `App`（http://www.scala-lang.org/api/current/#scala.App）和 `DelayedInit`（http://www.scala-lang.org/api/current/#scala.DelayedInit）特征
  使用这两个类型能很方便地实现 `main` 类型（入口类型），它们也是 Java 类中 `static main` 方法的同义词。不过由于它们有时候会导致奇怪的行为，因此我并不推荐使用它们。我会使用通用的、符合规范的 Scala 方法编写 `main` 方法。

- `scala.ref`（http://www.scala-lang.org/api/current/#scala.ref.package）
  对某些 Java 类型进行了封装，如 `WeakReference`，这是 `java.lang.ref.WeakReference` 的封装类。

- `scala.runtime`（http://www.scala-lang.org/api/current/#scala.runtime.package）
  用于实现类库的类型。

- `scala.util.hashing`（http://www.scala-lang.org/api/current/#scala.util.hashing.package）
  提供了多种散列算法。

# 欢迎阅读《Scala程序设计（第1版）》

一门编程语言能够流行起来是有一定原因的。有时候，某一平台的程序员会青睐于某一特定语言或平台提供商所建议的语言。大多数 Mac OS 开发者习惯使用 Objective-C，大多数 Windows 平台开发者使用 C++ 和 .NET 语言，而嵌入式系统开发者则使用 C 和 C++。

有时，语言能流行起来归功于其技术上的优势，这一优势能够使其变得时尚、让人着迷。C++、Java 和 Ruby 便曾引起程序员的狂热崇拜。

有时，语言会因为适应时代的需要而流行起来。Java 最初被视为一门能够用于编写基于浏览器的富客户端应用的完美语言。当面向对象编程变得主流时，Smalltalk 抓住了这一机遇。

现在，并发、异构型、永不停止的服务以及不断缩短的开发计划，使得业内对函数式编程越来越感兴趣。似乎面向对象编程的统治地位即将结束，而混合式编程范式将流行起来，甚至会变得不可或缺。

如今我们构建的应用大多是可靠、高性能、高并发的互联网或企业级应用程序，我们也希望会有一门通用编程语言适应这一要求，Scala 能将我们从其他语言的阵营中吸引过来，便是因为它具备了许多最理想的特性。

Scala 是一门多范式语言，同时支持面向对象和函数式编程。Scala 具有可扩展性，从小脚本到基于组件的大规模应用程序，Scala 均可胜任。Scala 是深奥的，它从全世界的计算机科学系中吸收了先进的思想。Scala 又是实用的，它的创建者 Martin Odesky 参与了多年的 Java 开发，能理解专业开发人员的需求。

Scala 简洁、优雅而又富有表现力的语法，以及提供的众多工具让我们为之着迷。本书力图阐明为什么这些特性会使 Scala 引人注目、不可或缺。

假如你是一位有经验的开发者，希望能快速全面地了解 Scala，那么这本书很适合你。你也许正在思考是否改用 Scala 或将其作为另一门补充语言；抑或你已经决定使用 Scala，需要学习并很好地掌握 Scala 的特性。无论是哪种情况，我们都希望能以一种平易近人的方式阐明这门强大的语言。

我们假设你已经很好地掌握了面向对象的编程，但并没有接触过函数式编程。我们认为你熟悉一门或多门其他的语言。我们对比了 Java、C#、Ruby 等语言的特性，如果你熟悉任何一种语言，会了解 Scala 中的相似特性，以及一些全新特性。

无论你是否具有面向对象或函数式编程的背景，都会了解到 Scala 如何优雅地融合这两种编范式，展示了它们的互补性。基于众多示例，你还能明白针对不同的设计问题如何以及何时应用 OOP 和 FP 技术。

最后，我们希望你也会为 Scala 着迷。即便 Scala 未能成为你日常使用的语言，无论你使用什么语言，我们也希望你能从 Scala 中洞察到些许知识。

## 排版约定

本书使用以下排版约定。

- 楷体

  表示新术语。

- 等宽字体（constant width）

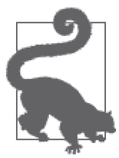
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**constant width bold**）

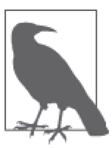
  表示应该由用户输入的命令或其他文本。

- 倾斜的等宽字体（*constant width italic*）

  表示应该由用户输入的值或根据上下文决定的值替换的文本。



该图标表示提示或建议。



该图标表示一般注记。



该图标表示警告或警示。

# 使用代码示例

本书就是要帮读者解决实际问题的。也许你需要在自己的程序或文档中用到本书中的代码。除非大段大段地使用，否则不必与我们联系取得授权。因此，使用本书中的几段代码写成一个程序不用向我们申请许可。但是，销售或者分发 O’Reilly 图书随附的代码光盘则必须事先获得授权。引用书中的代码来回答问题也无需我们授权。将大段的示例代码整合到你自己的产品文档中则必须经过许可。

使用我们的代码时，希望你能标明它的出处。出处一般要包含书名、作者、出版商和书号，例如：“*Programming Scala, Second Edition* by Dean Wampler and Alex Payne. Copyright 2015 Dean Wampler and Alex Payne, 978-1-491-94985-6.”

如果还有其他使用代码的情形需要与我们沟通，可以随时通过 permissions@oreilly.com 与我们联系。

## 获得示例代码

读者可以从 GitHub（https://github.com/deanwampler/prog-scala-2nd-ed-code-examples）下载代码示例，并把下载后的文件解压到指定位置。请阅读随示例发布的 README 文件，了解如何构建和使用这些示例。（第 1 章会对相关指令进行归纳说明。）

一些示例文件可以作为脚本，使用 `scala` 命令运行，而另外一些则必须编译成 class 文件，还有一些文件本身就包含了故意植入的错误，无法通过编译。为了表明文件类型，我采用了某种特定的文件命名方式。实际上，在学习 Scala 的过程中，你也能从文件内容中发现文件类型。在大多数情况下，本书示例文件遵循下列命名规范。

- *.scala

  这是 Scala 文件的标准文件扩展名，不过你无法从该扩展名中分辨该文件是必须使用 `scalac` 进行编译的源文件，还是可以直接使用 `scala` 运行的脚本文件，或者是本书特意植入了错误的无效代码文件。因此，本书示例代码中使用了 .scala 扩展名的文件必须单独经过编译才能使用，编译过程与编译 Java 代码相似。

- *.sc

  以 .sc 后缀结尾的文件可以作为脚本文件，使用 `scala` 命令运行。例如：`scala foo.sc` 命令会执行 `foo.sc` 脚本。你还可以在解释模式下启动 `scala`，并通过 `:load` 命令加载任意脚本文件。请注意，使用 .sc 对脚本进行命名并不是 Scala 社区的命名标准，不过由于 SBT 构建项目时会忽略 .sc 文件，因此我们在此处用其对脚本进行命名。与此同时，IDE 提供的 worksheet 新功能将 worksheet 文件命名为 .sc 文件，我们会在第 1 章中讨论这一功能。所以使用 .sc 对脚本命名是一个可以接受的、便利的命名方法。再次申明，通常情况下我们使用 .scala 扩展名为脚本文件和代码命名。

- *.scalaX 及 *.scX

  某些示例文件中特意植入了某些导致编译异常的错误。为了避免导致编译出错，这些文件使用了 .scalaX 或 .scX 扩展名。.scalaX 表示代码文件，而 .scX 则表示脚本文件。再次重申，.scalaX 和 .scX 扩展名并不是业内使用的扩展名。这些文件中也嵌入了一些注释，用于说明这些文件无法执行的原因。

## Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice

Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

# 联系我们

请把对本书的评价和发现的问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920033073.do

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

# 第2版致谢

我，Dean Wampler，在编写这一版书的过程中得到了 Typesafe 公司许多同事的指导和反馈。除此之外，一些审核了早期版本的人也给我提供了有价值的反馈，非常感谢他们。我要特别感谢 Ramnivas Laddad、Kevin Kilroy、Lutz Huehnken 和 Thomas Lockney，他们帮我审阅了本书的底稿。感谢我的老同事、老朋友 Jonas Bonér，感谢他为本书作序。

特别感谢 Ann，允许我将大量的个人时间花在本书的编写工作中，感谢你长期以来对我的包容，我爱你！

# 第1版致谢

我们在编写这本书的时候，一些朋友阅读了早期版本，并对本书的内容提出了大量好的意见，我们在此对这些朋友表示感谢。特别要感谢 Steve Jensen、Ramnivas Laddad、Marcel Molina、Bill Venners 和 Jonas Bonér，他们给本书提供了大量的反馈。

我们在 Safari 发布初稿以及在 http://programmingscala.com 网站上提供在线版本时，收到了大量的反馈。在此列举了提供反馈的读者（排名不分先后），对他们表示感谢。他们 是 Iulian Dragos、Nikolaj Lindberg、Matt Hellige、David Vydra、Ricky Clarkson、Alex Cruise、Josh Cronemeyer、Tyler Jennings、Alan Supynuk、Tony Hillerson、Roger Vaughn、Arbi Sookazian、Bruce Leidl、Daniel Sobral、Eder Andres Avila、Marek Kubica、Henrik Huttunen、Bhaskar Maddala、Ged Byrne、Derek Mahar、Geoffrey Wiseman、Peter Rawsthorne、Geoffrey Wiseman、Joe Bowbeer、Alexander Battisti、Rob Dickens、Tim MacEachern、Jason Harris、Steven Grady、Bob Follek、Ariel Ortiz、Parth Malwankar、Reid Hochstedler、Jason Zaugg、Jon Hanson、Mario Gleichmann、David Gates、Zef Hemel、Michael Yee、Marius Kreis、Martin Süsskraut、Javier Vegas、Tobias Hauth、Francesco Bochicchio、Stephen Duncan Jr.、Patrik Dudits、Jan Niehusmann、Bill Burdick、David Holbrook、Shalom Deitch、Jesper Nordenberg、Esa Laine、Gleb Frank、Simon Andersson、Patrik Dudits、Chris Lewis、Julian Howarth、Dirk Kuzemczak、Henri Gerrits、John Heintz、Stuart Roebuck 以及 Jungho Kim。还有很多读者也为本书提供了反馈，不过我们只知道他们的用户名，在此我们要向 Zack、JoshG、ewilligers、abcoates、brad、teto、pjcj、mkleint、dandoyon、Arek、rue、acangiano、vkelman、bryanl、Jeff、mbaxter、pjb3、kxen、hipertracker、ctran、Ram R.、cody、Nolan、Joshua、Ajay、Joe 表示感谢。除此之外，我们还要向不知道名字的贡献者表示感谢。如果名单中漏掉了谁，我们在此表示歉意！

Mike Loukides 是我们的编辑，他深知应该如何以温和的方式催促进度。他在我们写书的过程中为我提供了巨大的帮助。O'Reilly 出版社的其他员工也总能回答我们的问题，并帮助我们继续工作。

感谢 Jonas Bonér 为本书作序。Jonas 是我在"面向切面编程"（AOP，Aspect-Oriented Programming）委员会的老朋友、老同事。这些年来，他为 Java 社区做了很多前沿性的工作。现在他将精力投入到了改进 Scala 和发展 Scala 社区的工作中。

Bill Venners 很友善地评论了本书，我们将其置于封底处。他与 Martin Odersky、Lex Spoon 一起编写了第一本 Scala 图书《Scala 编程》。对于 Scala 开发人员而言，他是不可或缺的人物。Bill 同时还创造了令人惊叹的 ScalaTest 库。

除了 Jonas 和 Bill 之外，我们还从世界各地的开发人员处学到了很多，他们是 Debasish Ghosh、James Iry、Daniel Spiewak、David Pollack、Paul Snively、Ola Bini、Daniel Sobral、Josh Suereth、Robey Pointer、Nathan Hamblen、Jorge Ortiz，以及一些通过发表博文、参与论坛讨论以及私人会话给我提供帮助的朋友。

Dean 要向 Object Mentor 公司的同事表示感谢，他同时也要感谢许多客户端开发人员。他们激发了许多编程语言、软件设计以及业内实际问题的讨论。芝加哥地区 Scala 狂热者团

体（Chicago Area Scala Enthusiasts，CASE ）也为本书提供了很有价值的反馈及激励。

Alex 要向他在 Twitter 的同事表示感谢，他们对 Alex 的工作给予了鼓励，并在实际工作中很好地演示了 Scala 语言的能力。Alex 同时要对湾区 Scala 爱好者（Bay Area Scala Enthusiasts，BASE）表示感谢，他们的激情以及这个社团本身为他提供了帮助。

我们要特别感谢 Martin Odersky 和他的团队，感谢他们创造了 Scala。

# 第 1 章

# 零到六十：Scala简介

本章将简要说明 Scala 为何应该受到重视。之后我们将会深入学习 Scala，并动手编写一些代码。

## 1.1　为什么选择Scala

Scala 是一门满足现代软件工程师需求的语言；它是一门静态类型语言，支持混合范式；它也是一门运行在 JVM 之上的语言，语法简洁、优雅、灵活。Scala 拥有一套复杂的类型系统，Scala 方言既能用于编写简短的解释脚本，也能用于构建大型复杂系统。这些只是它的一部分特性，下面我们来详细说明。

- *运行在 JVM 和 JavaScript 之上的语言*

  Scala 不仅利用了 JVM 的高性能以及最优化性，Java 丰富的工具及类库生态系统也为其所用。不过 Scala 并不是只能运行在 JVM 之上！ Scala.js（http://www.scala-js.org）正在尝试将其迁移到 JavaScript 世界。

- *静态类型*

  在 Scala 语言中，*静态类型*（static typing）是构建健壮应用系统的一个工具。Scala 修正了 Java 类型系统中的一些缺陷，此外通过类型推演（type inference）也免除了大量的冗余代码。

- *混合式编程范式——面向对象编程*

  Scala 完全支持*面向对象编程*（OOP）。Scala 引入*特征*（trait）改进了 Java 的对象模型。trait 能通过使用*混合结构*（mixin composition）简洁地实现新的类型。在 Scala 中，一切都是对象，即使是数值类型。

- *混合式编程范式——函数式编程*

  Scala 完全支持*函数式编程*（FP），函数式编程已经被视为解决并发、大数据以及代码正确性问题的最佳工具。使用不可变值、被视为一等公民的函数、无副作用的函数、高阶函数以及函数集合，有助于编写出简洁、强大而又正确的代码。

- *复杂的类型系统*

  Scala 对 Java 类型系统进行了扩展，提供了更灵活的泛型以及一些有助于提高代码正确性的改进。通过使用类型推演，Scala 编写的代码能够和动态类型语言编写的代码一样精简。

- *简洁、优雅、灵活的语法*

  使用 Scala 之后，Java 中冗长的表达式不见了，取而代之的是简洁的 Scala 方言。Scala 提供了一些工具，这些工具可用于构建*领域特定语言*（DSL），以及对用户友好的 API 接口。

- *可扩展的架构*

  使用 Scala，你能编写出简短的解释性脚本，并将其粘合成大型的分布式应用。以下四种语言机制有助于提升系统的扩展性：1) 使用 trait 实现的混合结构；2) 抽象类型成员和泛型；3) 嵌套类；4) 显式*自类型*（self type）。

Scala 实际上是 Scalable Language 的缩写，意为可扩展的语言。Scala 的发音为 scah-lah，像意大利语中的 staircase（楼梯）。也就是说，两个 a 的发音是一样的。

早在 2001 年，Martin Odersky 便开始设计 Scala，并在 2004 年 1 月 20 日推出了第一个公开版本（参见 http://article.gmane.org/gmane.comp.lang.scala/17）。Martin 是瑞士洛桑联邦理工大学（EPFL）计算机与通信科学学院的一名教授。在就读研究生时，Martin 便加入了由 Niklaus Wirth[1] 领导的 PASCAL fame 项目组。Martin 曾任职于 Pizza 项目组，Pizza 是运行在 JVM 平台上早期的函数式语言。之后与 Haskell 语言设计者之一 Philip Wadler 一起转战 GJ。GJ 是一个原型系统，最终演变为 Java 泛型。Martin 还曾受雇于 Sun 公司，编写了 `javac` 的参考编译器，这套系统后来演化成了 JDK 中自带的 Java 编译器。

### 1.1.1 富有魅力的Scala

自从本书第 1 版出版之后，Scala 用户数量急剧上升，这也证实了我的观点：Scala 适应当前时代。当前我们会遇到很多技术挑战，如大数据、通过并发实现高扩展性、提供高可用并健壮的服务。Scala 语法简洁但却富有表现力，能够满足这些技术挑战。在享受 Scala 最先进的语言特性的同时，你还可以拥有成熟的 JVM、库以及生产工具给你带来的便利。

在那些需要努力才能成功的领域里，专家们往往都需要掌握复杂强大的工具和技术。也许掌握这些工具技能需要花费一些时间，但是掌握它们是你事业成功的关键，所以花费这些时间都是值得的。

注 1：PASCAL 之父。——译者注

我确信对于专家级开发者而言，Scala 就是这样一门语言。并不是所有的用户都能称得上是专家，而 Scala 却是属于技术专家的语言。Scala 包含丰富的语言特性，具有很好的性能，能够解决多种问题。虽然需要花一些时间才能掌握 Scala 语言，但是一旦你掌握了它，便不会被它束缚。

### 1.1.2　关于Java 8

自从 Java 5 引入泛型之后，再也没有哪次升级能比 Java 8 引入更多的特性了。现在可以使用真正的匿名函数了，我们称之为 Lambda。通过本书你将了解到这些匿名函数的巨大作用。Java 8 还改进了接口，允许为声明的方法提供默认实现。这一变化使得我们能够像使用混合结构那样使用接口，这也使接口变得更有用了，而 Scala 则是通过 trait 实现这种用法的。在 Java 8 推出之前，Scala 便已为 Java 提供了这两个被公认为 Java 8 最重要的新特性。现在是不是能说服自己切换到 Scala 了？

由于 Java 语言向后兼容的缘故，Scala 新增了一些改进，而 Java 也许永远不会包含。即便 Java 最终会拥有这些改进，那也需要漫长的等待。举例来说，较 Java 而言，Scala 能提供更为强大的类型推演、强大的模式匹配（pattern matching）和 `for` 推导式（for comprehension），善用模式匹配和 `for` 推导式能够极大地减少代码量以及类型耦合。随着深入学习，你会发现这些特性的巨大价值。

另外，一些组织对升级 JVM 设施抱有谨慎态度，这是可以理解的。对于他们而言，目前并不允许部署 Java 8 虚拟机。为了使用这些 Java 8 特性，这些组织可以在 Java 6 或 Java 7 的虚拟机上运行 Scala。

你也许因为当前使用 Java 8，就认为 Java 8 是最适合团队的选择。即便如此，本书仍然能给你传授一些有用的技术，而且这些技术可以运用在 Java 8 中。不过，我认为 Scala 具有一些额外的特性，能够让你觉得值得为之改变。

好吧，让我们开始吧！

## 1.2　安装Scala

为了能够尽可能快地安装并运行 Scala，本节将讲述如何安装命令行工具，使用这些工具便能运行本书列举的所有示例[2]。本书示例中的代码使用了 Scala 2.11.2 进行编写及编译。这也是编写本书时最新的版本。绝大多数代码无须修改便能运行在早期版本 2.10.4 上，而一些团队也仍在使用这一版本。

> 相较于 2.10，Scala 2.11 引入了一些新的特性，不过此次发布更侧重于整体性能的提升以及库的重构。Scala 2.10 与 2.9 版本相比，也引入了一些新的特性。也许你们部门正在使用其中的某一版本，而随着学习的深入，我们会讨论这些版本间最重要的差别。（参阅 http://docs.scala-lang.org/scala/2.11/ 了解 2.11 版本，参阅 http://www.scala-lang.org/download/2.10.4.html#Release_Notes 了解 2.10 版本。）

注 2：第 21 章会详细讲解这些工具。

安装步骤如下。

- 安装 Java

  针对 Scala 2.12 之前的版本，你可以选择 Java 6、7、8 三个版本，在安装 Scala 之前，你必须确认你的电脑上已经安装了 Java。（Scala 2.12 计划于 2016 年年初发布，该版本将只支持 Java 8。）假如你需要安装 Java，请登录 Oracle 的网站（http://www.oracle.com/technetwork/java/javase/downloads/index.html），遵循指示安装完整的 Java 开发工具包（JDK）。

- 安装 SBT

  请遵循 scala-sbt.org（http://www.scala-sbt.org/release/tutorial/Setup.html）网页上的指示安装 SBT，它是业内公认的构建工具。安装完成后，便可以在 Linux、OS X 终端和 Windows 命令窗口中运行 `sbt` 命令。（你也可以选择其他的构建工具，21.2.2 节将介绍这些工具。）

- 获取本书源代码

  本书前言中描述了如何下载示例代码。压缩包可以解压到你电脑中的任何文件夹。

- 运行 SBT

  打开 shell 或命令行窗口，进入示例代码解压后的目录，敲入命令 **`sbt test`**，该命令会下载所有的依赖项，包括 Scala 编译器及第三方库，请确保网络连接正常，并耐心等待该命令执行。下载完毕后，`sbt` 会编译代码并运行单元测试。此时你能看到很多的输出信息，该命令最后会输出 success 信息。再次运行 `sbt test` 命令，由于该命令不需要执行任何事情，你会发现命令很快就结束了。

祝贺你！你已经真正开始了 Scala 的学习。不过，你也许会想安装其他一些有用的工具。



在学习本书的大多数时候，通过使用 SBT，你便能使用其他工具。SBT 会自动下载指定版本的 Scala 编译器、标准库以及需要的第三方资源。

不使用 SBT，也能很方便地单独下载 Scala 工具。我们会提供一些 SBT 外使用 Scala 的例子。

请遵循 Scala 官方网站（http://www.scala-lang.org）中的链接安装 Scala，还可以选择安装 Scaladoc。Scaladoc 是 Scala 版的 Javadoc（在 Scala 2.11 中，Scala 库和 Scaladoc 被切分为许多较小的库）。你也可以在线查阅 Scaladoc（http://www.scala-lang.org/api/current）。为了方便你使用，本书中出现的 Scala 库中的类型，大部分都附上了连接到 Scaladoc 页面的链接。

Scaladoc 在页面左侧类型列表上面提供了搜索栏，这有助于快速查找类型。同时，每个类型的入口处都提供了一个指向 Scala GitHub 库中对应代码的链接（https://github.com/scala/scala），这能很好地帮助用户学习这些库的实现。这个链接位于类型概述讨论的底部，链接所在行标注着 Source 字样。

你可以选用任何文本编辑器或 IDE 来处理这些示例，也可以为这些主流编辑器或 IDE 安装 Scala 支持插件。具体方法，请参见 21.3 节。通常而言，访问你所青睐的编辑器的社区，能最及时地发现 Scala 相关的支持信息。

### 1.2.1 使用SBT

21.2.1 节将介绍 SBT 是如何工作的。下面，我们介绍当前需要掌握的一些基本指示。

当你启动 `sbt` 命令时，假如不指定任何任务，SBT 将启动一个交互式 REPL（REPL 是 Read、Eval、Print、Loop 的简写，代表了“读取 – 求值 – 打印 – 循环”）。下面我们将运行该命令，并尝试运行一些可用的任务。

下面列举的代码中，`$` 表示 shell 命令提示符（如 `bash` 命令提示符），你可以在该提示符下运行 `sbt` 命令；`>` 是 SBT 默认的交互提示符，可以在 `#` 符号后编写 `sbt` 注释。你可以以任意顺序输入下面列举的大多数 `sbt` 命令。

```
$ sbt
> help          # 描述命令
> tasks         # 显示最常用的、当前可用的任务
> tasks -V      # 显示所有的可用任务
> compile       # 增量编译代码
> test          # 增量编译代码,并执行测试
> clean         # 删除所有已经编译好的构建
> ~test         # 一旦有文件保存,执行增量编译并运行测试
                # 适用于任何使用了~前缀的命令
> console       # 运行Scala REPL
> run           # 执行项目的某一主程序
> show x        # 显示变量X的定义
> eclipse       # 生成Eclipse项目文件
> exit          # 退出REPL(也可以通过control-d的方式退出)
```

为了能编译更新后的代码并运行对应测试，我通常会执行 `~test` 命令。SBT 使用了增量的编译器和测试执行器，因此每次执行时不用等待完全构建所需时间。假如你希望执行其他任务或退出 `sbt`，只需要按一下回车键即可。

假如你使用安装了 Scala 插件的 Eclipse 进行开发，便能很方便地执行 `eclipse` 任务。运行 `eclipse` 任务将生成对应的项目文件，这些生成的代码作为 Eclipse 项目文件进行加载。如果你想使用 Eclipse 来处理示例代码，请执行 `eclipse` 任务。

假如你使用最近发布的 Scala 插件 IntelliJ IDEA 进行开发，直接导入 SBT 项目文件便能生成 IntelliJ 项目。

Scala 中已经包含了 REPL 环境，你可以执行 `console` 命令启动该环境。如果你希望在 REPL 环境下运行本书中的代码示例，那么通常情况下，你首先需要运行 `console` 命令：

```
$ sbt
> console
[info] Updating {file:/.../prog-scala-2nd-ed/}prog-scala-2nd-ed...
[info] ...
[info] Done updating.
[info] Compiling ...
```

```
[info] Starting scala interpreter...
[info]
Welcome to Scala version 2.11.2 (Java HotSpot(TM) 64-Bit Server VM, Java ...).
Type in expressions to have them evaluated .
Type :help for more information.

scala> 1 + 2
res0: Int = 3

scala> :quit
```

此处省去若干输出，与 SBT REPL 一样，你也可以使用 Ctrl-D 退出系统。

运行 `console` 时，SBT 首先会构建项目，并通过设置 `CLASSPATH` 使该项目可用。因此，你也可以使用 REPL 编写代码进行试验。

> 使用 Scala REPL 能有效地对你编写的代码进行试验，也可以通过 REPL 来学习 API，即便是 Java API 亦可。在 SBT 上使用 `console` 任务执行代码时，`console` 任务会很体贴地为你在 classpath 中添加项目依赖项以及编译后的项目代码。

## 1.2.2　执行Scala命令行工具

如果你单独安装了 Scala 命令行工具，会发现与 Java 编译器 `javac` 相似，Scala 编译器叫作 `scalac`。我们会使用 SBT 执行编译工作，而不会直接使用 `scalac`。不过如果你曾运行过 `javac` 命令，会发现 `scalac` 语法也很直接。

在命令行窗口中运行 `-version` 命令，便可查看到当前运行的 `scalac` 版本以及命令行参数帮助信息。与之前一样，在 `$` 提示符后输入文本。之后生成的文本便是命令输出。

```
$ scalac -version
Scala compiler version 2.11.2 -- Copyright 2002-2013, LAMP/EPFL
$ scalac -help
Usage: scalac <options> <source files>
where possible standard options include:
  -Dproperty=value       Pass -Dproperty=value directly to the runtime system.
  -J<flag>               Pass <flag> directly to the runtime system.
  -P:<plugin>:<opt>      Pass an option to a plugin
  ...
```

与之类似，执行下列 `scala` 命令也可以查看 Scala 版本及命令参数帮助。

```
$ scala -version
Scala code runner version 2.11.2 -- Copyright 2002-2013, LAMP/EPFL
$ scala -help
Usage: scala <options> [<script|class|object|jar> <arguments>]
   or  scala -help

All options to scalac (see scalac -help) are also allowed.
...
```

有时我们会使用 scala 来运行 Scala“脚本”文件，而 java 命令行却没有提供类似的功能。下面将要执行的脚本来源于我们的示例代码：

```
// src/main/scala/progscala2/introscala/upper1.sc

class Upper {
  def upper(strings: String*): Seq[String] = {
    strings.map((s:String) => s.toUpperCase())
  }
}

val up = new Upper
println(up.upper("Hello", "World!"))
```

我们将调用 scala 命令执行该脚本。也请读者尝试运行该示例。上述代码使用的文件路径适用于 Linux 和 Mac OS X 系统。我假设，当前的工作目录位于代码示例所在的根目录。如果使用 Windows 系统，请在路径中使用反斜杠。

```
$ scala src/main/scala/progscala2/introscala/upper1.sc
ArrayBuffer(HELLO, WORLD!)
```

现在我们终于满足了编程图书或向导的一条不成文的规定：第一个程序必须打印“Hello World!”。

最后提一下，执行 scala 命令时，如果未指定主程序或脚本文件，那么 scala 将进入 REPL 模式，这与在 sbt 中运行 console 命令类似。（不过，运行 scala 时的 classpath 与执行 console 任务的 classpath 不同。）下面列出的 REPL 会话中讲解了一些有用的命令。（如果你未独立安装 Scala，在 sbt 中执行 console 任务也能进入 Scala REPL 环境）。此时，REPL 提示符是 scala>（此处省略了一些输出信息）。

```
$ scala
Welcome to Scala version 2.11.2 (Java HotSpot(TM)...).
Type in expressions to have them evaluated.
Type :help for more information.

scala> :help
All commands can be abbreviated, e.g. :he instead of :help.
:cp <path>                add a jar or directory to the classpath
:edit <id>|<line>         edit history
:help [command]           print this summary or command-specific help
:history [num]            show the history (optional num is commands to show)
... 其他消息

scala> val s = "Hello, World!"
s: String = Hello, World!

scala> println("Hello, World!")
Hello, World!

scala> 1 + 2
res3: Int = 3

scala> s.con<tab>
```

```
concat   contains   contentEquals

scala> s.contains("el")
res4: Boolean = true

scala> :quit
$    #返回shell提示符
```

我们为变量 s 赋予了 string 值 "Hello，World!"，通过使用 val 关键字，我们将变量 s 声明成不可变值。println 函数（http://www.scala-lang.org/api/current/index.html#scala.Console$）将在控制台中打印一个字符串，并会在字符串结尾处打印换行符。

println 函数与 Java 中的 System.out.println（http://docs.oracle.com/javase/8/docs/api/java/lang/System.html）作用一致。同样，Scala 也使用了 Java 提供的 String 类型（http://docs.oracle.com/javase/8/docs/api/java/lang/String.html）。

接下来，请注意我们要将两个数字相加，由于我们并未将运算的结果赋予任何一个变量，因此 REPL 帮我们将变量命名为 res3，我们可以在随后的表达式中运用该变量。

REPL 支持 tab 补全。例子中显示输入命令 s.con<tab> 表示的是在 s.con 后输入 tab 符。REPL 将列出一组可能会被调用的方法名。在本例中表达式最后调用了 contains 方法。

最后，调用 :quit 命令退出 REPL。也可以使用 Ctrl-D 退出。

接下来，我们将看到更多 REPL 命令，在 21.1 节中，我们将更深入地探索 REPL 的各个命令。

### 1.2.3 在IDE中运行Scala REPL

下面我们将讨论另外一种执行 REPL 的方式。特别是当你使用 Eclipse、IntelliJ IDEA 或 NetBeans 时，这种方式会更加实用。Eclipse 和 IDEA 支持 worksheet 功能，当你编辑 Scala 代码时，感觉不到它与正常地编辑编译代码或脚本代码有什么区别。不过一旦将该文件保存，代码便会立刻被执行。因此，假如你需要修改并重新运行重要的代码片段，使用这种开发方式比使用 REPL 更为方便。NetBeans 也提供了一种类似的*交互式控制台*功能。

假如你想要使用上述的某个 IDE，可以参考 21.3 节，掌握 Scala 插件、worksheet 以及交互式控制台的相关信息。

## 1.3 使用Scala

在本章的剩余篇幅和之后的两章中，我们将对 Scala 的一些特性进行快速讲解。在学习这些内容时会涉及一些语言细节，这些细节仅用于理解这些内容，更多的细节会在后续章节中提供。你可以将这几章内容视为 Scala 语法入门书，并从中感受 Scala 编程的魅力。

> 当提到某一 Scala 库类型时，我们可以阅读 Scaladoc 中的相关信息进行学习。如果你想访问当前版本的 Scala 对应的 Scaladoc 文档，请查看 http://www.scala-lang.org/api/current/。请注意，左侧类型列表区域的上方有一搜索栏，应用该搜索栏能很方便地快速查找类型。与 Javadoc 不同，Scaladoc 按照 package 来排列类型，而不是按照字母顺序全部列出。

本书多数情况下会使用 Scala REPL，因此我们在这儿再温习一遍运行 REPL 的三种方式。你可以不指定脚本或 `main` 参数直接输入 `scala` 命令，也可以使用 SBT `console` 命令，还可以在那些流行的 IDE 中使用 worksheet 特性。

假如你不想使用任何 IDE，我建议你尽量使用 SBT，尤其是当你的工作固定在某一特定项目时。本书也将使用 SBT 进行讲解，这些操作步骤同样适用于直接运行 `scala` 命令或者在 IDE 中创建 worksheet 的情况。请自行选择开发工具。事实上，即便你青睐于使用 IDE，我还是希望你能尝试在命令行窗口运行一下 SBT，了解 SBT 环境。我个人很少使用 IDE，不过是否选择 IDE 只是个人的偏好罢了。

打开 shell 窗口，切换到代码示例所在的根文件夹并运行 `sbt`。在 `>` 提示符后输入 **`console`**。从现在开始，本书将省略关于 `sbt` 和 `scala` 输出的 一些“程序化”的语句。

在 `scala>` 提示符中输入下列两行：

```
scala> val book = "Programming Scala"
book: java.lang.String = Programming Scala

scala> println(book)
Programming Scala
```

第一行代码中的 `val` 关键字用于声明不变变量 `book`。可变数据是错误之源，因此我推荐使用不变值。

请注意，解释器返回值列出了 `book` 变量的类型和数值。Scala 从字面量 `"Programming Scala"` 中推导出 `book` 属于 `java.lang.String` 类型（http://docs.oracle.com/javase/8/docs/api/java/lang/String.html）。

显示类型信息或在声明中显式指明类型信息时，这些类型标注紧随冒号，出现在相关项之后。为什么 Scala 不遵循 Java 的习惯呢？ Scala 常常能推导出类型信息，因此，我们在代码中总是看不到显式的类型标注。如果代码中省略了冒号和类型标注信息，那么与 Java 的类型习惯相比，`item：type` 这一模式更有助于编译器正确地分析代码。

一般来说，当 Scala 语法与 Java 语法存在差异时，通常都会有一个充分的理由。比如说，Scala 支持了一个新的特性，而这个特性很难使用 Java 的语法表达出来。

> REPL 中显示了类型信息，这有助于学习 Scala 是如何为特定表达式推导类型的。透过这个例子，可以了解到 REPL 提供了哪些功能。

仅使用 REPL 来编辑或提交大型的示例代码会比较枯燥，而使用文本编辑器或 IDE 来编写 Scala 脚本则会方便得多。编写完成之后，你可以执行脚本，也可以复制粘贴大段代码再执行。

我们再回顾一下之前编写的 upper1.sc 文件。

```
// src/main/scala/progscala2/introscala/upper1.sc

class Upper {
  def upper(strings: String*): Seq[String] = {
    strings.map((s:String) => s.toUpperCase())
  }
}

val up = new Upper
println(up.upper("Hello", "World!"))
```

本书的下载示例压缩包中的每个示例的第一行均为注释，该注释列出了示例文件在压缩包中的路径。Scala 遵循 Java、C#、C 等语言的注释规则，`// comment` 只能作用到本行行尾，而 `/* comment */` 则可以跨行。

我们再回顾一下前言中的内容。依照命名规范，脚本文件的扩展名为 .sc，而编译后的文件的扩展名为 .scala，这一命名规范仅适用于本书。通常，脚本文件往往也使用 .scala 扩展名。不过如果使用 SBT 构建项目，SBT 会尝试编译这些以 scala 命名的文件，而这些脚本文件却无法编译（我们稍后便会讲到这些）。

我们首先运行该脚本，具体代码细节稍后讨论。启动 `sbt` 并执行 `console` 命令以开启 Scala 环境。然后使用 `:load` 命令加载（编译并运行）文件：

```
scala> :load src/main/scala/progscala2/introscala/upper1.sc
Loading src/main/scala/progscala2/introscala/upper1.sc...
defined class Upper
up: Upper = Upper@4ef506bf    // 调用Java的Object.toString方法。
ArrayBuffer(HELLO, WORLD!)
```

上述脚本中，只有最后一行才是 `println` 命令的输出，其他行则是 REPL 提供的一些反馈信息。

那么这些脚本为什么无法编译呢？脚本设计的初衷是为了简化代码，无须将声明（变量和函数）封装在对象中便是一种简化。而将 Java 和 Scala 代码编译后，声明必须封装在对象中（这是 JVM 字节码的需求）。`scala` 命令通过一个聪明的技巧解决了冲突：将脚本封装在一个你看不到的匿名对象中。

假如你的确希望能将脚本文件编译为 JVM 的字节码（一组 .class 文件），可以在 `scalac` 命令中传入 `-Xscript <object>` 参数，`<object>` 表示你所选中的 main 类，它是生成的 Java 应用程序的入口点。

```
$ scalac -Xscript Upper1 src/main/scala/progscala2/introscala/upper1.sc
$ scala Upper1
ArrayBuffer(HELLO, WORLD!)
```

执行完毕后检查当前文件夹，你会发现一些命名方式有趣的 .class 文件。（提示：一些匿名函数也被转换成了对象！）我们稍后会再讨论这些名字，Upper1.class 文件中包含了主程序，我们将使用 `javap` 和 Scala 对应工具 `scalap`，对该文件实施逆向工程！

```
$ javap -cp . Upper1
Compiled from "upper1.sc"
public final class Upper1 {
```

```
  public static void main(java.lang.String[]);
}
$ scalap -cp . Upper1
object Upper1 extends scala.AnyRef {
  def this() = { /* compiled code */ }
  def main(argv : scala.Array[scala.Predef.String]) : scala.Unit =
    { /* compiled code */ }
}
```

最后，我们将对代码本身进行讨论，代码如下：

```
// src/main/scala/progscala2/introscala/upper1.sc

class Upper {
  def upper(strings: String*): Seq[String] = {
    strings.map((s:String) => s.toUpperCase())
  }
}

val up = new Upper
println(up.upper("Hello", "World!"))
```

Upper 类中的 upper 方法将输入字符串转换成大写字符串，并返回一个包含这些字符串的 Seq（Seq 表示“序列”，http://www.scala-lang.org/api/current/index.html#scala.collection.Seq）对象。最后两行代码创建了 Upper 对象的一个实例，并调用这一实例将字符串“Hello”和“World!”转换为大写字符串，并最终打印出产生的 Seq 对象。

在 Scala 中定义类时需要输入 class 关键字，整个类定义体包含在最外层的一对大括号中（{…}）。事实上，这个类定义体同样也是这个类的主构造函数。假如需要将参数传递给这个构造函数，就要在类名 Upper 之后输入参数列表。

下面这小段代码声明了一个方法：

```
def upper(strings: String*): Seq[String] = ...
```

定义方法时需要先输入 def 关键字，之后输入方法名称以及可选的参数列表。再输入可选的返回类型（有时候，Scala 能够推导出返回类型），返回类型由冒号加类型表示。最后使用等于号（=）将方法签名和方法体分隔开。

实际上，圆括号中的参数列表代表了变长的 String 类型参数列表，修饰 strings 参数的 String 类型后面的 * 号指明了这一点。也就是说，你可以传递任意多的字符串（也可以传递空列表），而这些字符串由逗号分隔。在这个方法中，strings 参数的类型实际上是 WrapppedArray（http://www.scala-lang.org/api/current/index.html#scala.collection.mutable.WrappedArray），该类型对 Java 数组进行了封装。

参数列表后列出了该方法的返回类型 Seq[String]，Seq（代表 Sequence）是集合的一种抽象，你可以依照固定的顺序（不同于遍历 Set 和 Map 对象那样的随机顺序和未定义顺序，遍历那类容器无法保证遍历顺序）遍历这类结合抽象。实际上，该方法返回的类型是 scala.collection.mutable.ArrayBuffer（http://www.scala-lang.org/api/current/#scala.collection.mutable.ArrayBuffer），不过绝大多数情况下，调用者无须了解这点。

值得一提的是，`Seq` 是一个参数化类型，就好象 Java 中的泛型类型。`Seq` 代表着“某类事物的序列”，上面代码中的 `Seq` 表示的是一个字符串序列。请注意，Scala 使用方括号（`[…]`）表示参数类型，而 Java 使用角括号（`<…>`）。

> Scala 的标识符，如方法名和变量名，中允许出现尖括号，例如定义“小于”方法时，该方法常被命名为 `<`，这在 Scala 语言中是允许的，而 Java 则不允许标识符中出现这样的字符。因此，为了避免出现歧义，Scala 使用方括号而不是尖括号表示参数化类型，并且不允许在标识符中使用方括号。

`upper` 方法的定义体出现在等号（`=`）之后。为什么使用等号呢？而不像 Java 那样，使用花括号表示方法体呢？

避免歧义是原因之一。当你在代码中省略分号时，Scala 能够推断出来。在大多数时候，Scala 能够推导出方法的返回类型。假如方法不接受任何参数，你还可以在方法定义中省略参数列表。

使用等号也强调了函数式编程的一个准则：值和函数是高度对齐的概念。正如我们所看到的那样，函数可以作为参数传递给其他函数，也能够返回函数，还能被赋给某一变量。这与对象的行为是一致的。

最后提一下，假如方法体仅包含一个表达式，那么 Scala 允许你省略花括号。所以说，使用等号能够避免可能的解析歧义。

函数方法体中对字符串集合调用了 `map` 方法（http://www.scala-lang.org/api/current/index.html#scala.collection.TraversableLike），`map` 方法的输入参数为*函数字面量*（function literal）。而这些函数字面量便是“匿名”函数。在其他语言中，它们也被称为 Lambda、*闭包*（closure）、*块*（block）或*过程*（proc）。Java 8 最终也提供了真正的匿名方法 Lambda。但 Java 8 之前，你只能通过接口实现的方式实现匿名方法，我们通常会在接口中定义一个匿名的内部类，并在内部类中声明执行真正工作的方法。因此，即便是在 Java 8 之前，你也能够实现匿名函数的功能：通过传入某些嵌套行为，将外部行为参数化。不过这些繁琐的语法着实损害并掩盖了匿名方法这门技术的优势。

在这个示例中，我们向 `map` 方法传递了下列函数字面量：

```
(s:String) => s.toUpperCase()
```

此函数字面量的参数表中只包含了一个字符串参数 `s`。它的函数体位于箭头 `=>` 之后（UTF8 也允许使用 `⇒`）。该函数体调用了 `s` 的 `UpperCase()` 方法。此次调用的返回值会自动被这个函数字面量返回。在 Scala 中，函数或方法中把最后一条表达式的返回值作为自己的返回值。尽管 Scala 中存在 `return` 关键字，但只能在方法中使用，上面这样的匿名函数则不允许使用。事实上，方法中也很少用到这个关键字。

> **方法和函数**
>
> 对于大多数的面向对象编程语言而言，方法指的是类或对象中定义的函数。当调用方法时，方法中的 this 引用会隐性地指向某一对象。当然，在大多数的 OOP 语言中，方法调用的语法通常是 this.method_name(other_args)。本书中的"方法"也满足这一常用规范。我们提到的"函数"尽管不是方法，但在某些时候通常会将方法也归入函数。当前上下文能够认清它们的区别。
>
> upper1.sc 中表达式 (s:String) => s.toUpperCase() 便是一个函数，它并不是方法。

我们对序列对象 strings 调用了 map 方法，该方法会把每个字符串依次传递给函数字面量，并将函数字面量返回的值组成一个新的集合。举个例子，假如在原先的列表中有五个元素，那么新生成的列表也将包含五个元素。

继续上面的示例，为了进一步练习代码，我们会创建一个新的 Upper 实例并将它赋给变量 up。与 Java、C# 等类似语言一样，new Upper 语法将创建一个新的实例。由于主构造函数并不接受任何参数，因此并不需要传递参数列表。通过 val 关键字，up 参数被声明为只读值。up 的行为与 Java 中的 final 变量相似。

最后，我们调用 upper 方法，并使用 println(…) 方法打印结果。

我们可以进一步简化代码，请思考下面更简洁的版本。

```
// src/main/scala/progscala2/introscala/upper2.sc

object Upper {
  def upper(strings: String*) = strings.map(_.toUpperCase())
}

println(Upper.upper("Hello", "World!"))
```

这段代码同样实现了相同的功能，但使用的字符却少了三分之一。

在第一行中，Upper 被声明为单例对象，Scala 将单例模式视为本语言的第一等级成员。尽管我们声明了一个类，不过 Scala 运行时只会创建 Upper 的一个实例。也就是说，你无法通过 new 创建 Upper 对象。就好像 Java 使用静态类型一样，其他语言使用*类成员*（class-level member），Scala 则使用对象进行处理。由于 Upper 中并不包含状态信息，所以我们此处的确不需要多个实例，使用单例便能满足需求。

单例模式具有一些弊端，也因此常被指责。例如在那些需要将对象值进行 double 的单元测试中，如果使用了单例对象，便很难替换测试值。而且如果对一个实例执行所有的计算，会引发线程安全和性能的问题。不过正如静态方法或静态值有时适用于 Java 这样的语言一样，单例有时候在 Scala 中也是适用的。上述示例便是一个证明，由于无须维护状态而且对象也不需要与外界交互，单例模式适用于上述示例。因此，使用 Upper 对象时我们没有必要考虑测试双倍值的问题，也没有必要担心线程安全。

> Scala 为什么不支持静态类型呢？与那些允许静态成员（或类似结构）的语言相比，Scala 更信奉万物皆应为对象。相较于混入了静态成员和实例成员的语言，采用对象结构的 Scala 更坚定地贯彻了这一方针。回想一下，Java 的静态方法和静态域并未绑定到类型的实际实例中，而 Scala 的对象则是某一类型的单例。

第二行中 `upper` 的实现同样简洁。尽管 Scala 无法推断出方法的参数类型，却常常能够推断出方法的返回类型，因此我们在此省略返回类型的显式声明。同时，由于方法体中仅包含了一句表达式，我们可以省略括号，并在一行内完成整个方法的定义。除了能提示读者之外，方法体之前的等号也告诉编译器方法体的起始位置。

Scala 为什么无法推导出方法参数类型呢？理论上类型推理算法执行了局部类型推导，这意味着该推导无法作用于整个程序全局，而只能局限在某一特定域内。因此，尽管无法分辨出参数所必须使用的类型，但由于能够查看整个函数体，Scala 大多数情况下却能推导出方法的返回值类型。递归函数是个例外，由于它的执行域超越了函数体的范围，因此必须声明返回类型。

任何时候，参数列表中的返回类型都为读者提供了有用信息。仅仅是因为 Scala 能推导出函数的返回类型，我们就放弃为读者提供返回类型信息吗？对于简单的函数而言，读者能够很清楚地发现返回类型，显式列出的返回类型也许还不是特别重要。不过有时候由于 bug 或某些特定输入或函数体中的某些表达式所触发的某些微妙行为，推导出的类型可能并不是我们所期望的类型。显式返回类型代表了你所期望的返回类型，它们同时还为读者提供了有用信息，因此我推荐添加返回类型，而不要省略它们。这尤其适用于公有 API。

我们对函数字面量进行了进一步的简化，之前我们的代码如下：

```
(s:String) => s.toUpperCase()
```

我们将其简化为下列表达式：

```
_.toUpperCase()
```

`map` 方法接受单一函数参数，而单一函数也只接受单一参数。在这种情况下，函数体只使用一次该参数，所以我们使用占位符 _ 来替代命名参数。也就是说：_ 起到了匿名参数的作用，在调用 `toUpperCase` 方法之前，_ 将被字符串替换。Scala 同时也为我们推断出了该变量的类型为 `String` 类型。

最后一行代码中，由于使用了对象而不是类，此次调用变得更加简单。无须通过 `new Upper` 代码创建实例，我们只需直接调用 `Upper` 对象的 `upper` 方法。调用语法与调用 Java 类静态方法时的语法一样。

最后，Scala 会自动加载一些像 `println`（http://www.scala-lang.org/api/current/index.html#scala.Console$）这样的 I/O 方法，`println` 方法实际是 `scala` 包（http://www.scala-lang.org/api/current/scala/package.html）中 `Console` 对象（http://www.scala-lang.org/api/current/index.html#scala.Console$）的一个方法。与 Java 中的包一样，Scala 通过包提供"命名空间"并界定作用域。

因此，使用 `println` 方法时，我们无需调用 `scala.Console.println` 方法（http://www.scala-lang.org/api/current/index.html#scala.Console$），直接输入 `println` 即可。`println` 方法只是众多被自动加载的方法和类型中的一员，有一个叫作 `Predef` 的库对象（http://www.scala-lang.org/api/current/index.html#scala.Predef$）对这些自动加载的方法和类型进行定义。

我们再进行一次重构，把这个脚本转化成编译好的一个命令行工具。也就是说，我们将创建一个包含了 `main` 方法的更为经典的 JVM 应用程序。

```
// src/main/scala/progscala2/introscala/upper1.scala
package progscala2.introscala

object Upper {
  def main(args: Array[String]) = {
    args.map(_.toUpperCase()).foreach(printf("%s ",_))
    println("")
  }
}
```

回顾一下前面的内容，如果代码具有 .scala 扩展名，那就表示我们会使用 `scalac` 编译它。现在 `upper` 方法被改名成了 `main` 方法。由于 `Upper` 是一个对象，`main` 方法就像是 Java 类的静态 `main` 方法一样。它就是 `Upper` 应用的入口点。

在 Scala 中，`main` 方法必须为对象方法。（在 Java 中，`main` 方法必须是类静态方法。）应用程序的命令行参数将作为一组字符串传递给 `main` 方法。举例来说，输入参数是 `args: Array[String]`。

upper1.scala 文件中的第一行代码定义了名为 `introscala` 的包，用于装载所定义的类型。在 `Upper.main` 方法中的表达式使用了 `map` 方法的简写形式，这与我们之前代码中出现的简写形式一致。

```
args.map(_.toUpperCase())...
```

`map` 方法会返回一个新的集合。对该集合我们将使用 `foreach` 方法进行遍历。我们向 `foreach` 方法中传递另一个使用了 `_` 占位符的函数字面量。在这段代码中，集合中的每一个字符串都将作为参数传递给 `scala.Console.printf` 方法（http://www.scala-lang.org/api/current/index.html#scala.Console$），该方法也是 `Predef` 对象导入的方法，它会接受代表格式的字符串参数以及一组将嵌入到格式字符串的参数。

```
args.map(_.toUpperCase()).foreach(printf("%s ",_))
```

在此澄清一下，上述代码有两处使用了 `_`，这两个 `_` 分别位于不同的作用域中，彼此之间没有任何关联。

你需要花一些时间才能掌握这样的链式函数以及函数字面量中的一些简写方式，不过一旦熟悉了它们，你便能应用它们编写出可读性强、简洁强大的代码，这些代码能最大程度地避免使用临时变量和其他一些样板代码。如果你是一名 Java 程序员，可以想象一下使用早于 Java 8 的 Java 版本编写代码，这时你需要使用匿名内部类才能实现相同的功能。

main 方法的最后一行在输出中增加了一个最终换行符。

为了运行代码，你必须首先使用 scalac，将代码编译成一个能在 JVM 下运行的 .class 文件（下文中的 $ 代表命令提示符）。

```
$ scalac src/main/scala/progscala2/introscala/upper1.scala
```

现在，你应该会看到一个名为 progscala2/introscala 的新文件夹，该文件夹里包含了一些 .class 文件，Upper.class 便是其中的一个文件。Scala 生成的代码必须满足 JVM 字节代码的合法性要求，文件夹目录必须与包结构吻合是要求之一。

Java 在源代码级也遵循这一规定，Scala 则要更灵活一些。请注意，在我们下载的代码示例中，文件 Upper.class 位于一个叫作 IntroScala 的文件夹中，这与它的包名并不一致。Java 同时要求必须为每一个最顶层类创建一个单独的文件，而 Scala 则允许在文件中创建任意多个类型。虽然开发 Scala 代码可以不用遵循 Java 关于源代码目录结构的规范（源代码目录结构应吻合包结构，而且为每个顶层类创建一个单独的文件），不过一些开发团队依然遵循这些规范，这主要因为他们熟悉这些 Java 规范，而且遵循这些规范有利于追踪代码位置。

现在，你可以输入任意长度的字符串参数并执行命令，如下所示：

```
$ scala -cp . progscala2.introscala.Upper Hello World!
HELLO WORLD!
```

我们通过选项 -cp . 将当前目录添加到查询类路径（classpath）中，不过本示例其实并不需要该选项。

请尝试使用其他输入参数来执行程序。另外，你可以查看 progscala2/introscala 文件夹中还有哪些其他的类文件，像之前例子那样使用 javap 或 scalap 命令查看这些类中包含了什么定义。

最后，由于 SBT 会帮助我们编译文件，我们实际上并不需要手动编译这些文件。在 SBT 提示符下，我们可以使用下列命令运行程序。

```
> run-main progscala2.introscala.Upper Hello World!
```

使用 scala 命令运行程序时，我们需要指明 SBT 生成的类文件的正确路径。

```
$ scala -cp target/scala-2.11/classes progscala2.introscala.Upper Hello World!
HELLO WORLD!
```

### 解释运行 Scala 与编译运行 Scala

概括地说，假如在命令行输入 scala 命令时不指定文件参数，REPL 将启动。在 REPL 中输入的命令、表达式和语句都会被直接执行。假如输入 scala 命令时指定 Scala 源文件，scala 命令将会以脚本的形式编译并运行文件。另外，假如你提供了 JAR 文件或是一个定义了 main 方法的类文件，scala 会像 Java 命令那样执行该文件。

我们接下来对这些代码再进行最后一次重构：

```
// src/main/scala/progscala2/introscala/upper2.scala
package progscala2.introscala

object Upper2 {
  def main(args: Array[String]) = {
    val output = args.map(_.toUpperCase()).mkString(" ")
    println(output)
  }
}
```

将输入参数映射为大写格式字符串之后，我们并没有使用 `foreach` 方法迭代并依次打印每个词，而是通过一个更便利的集合方法生成字符串。`mkString` 方法（http://www.scala-lang.org/api/current/index.html#scala.collection.TraversableOnce）只接受一个输入参数，该参数指定了集合元素间的分隔符。另外一个 `mkString` 方法（重构版本）则接受三个参数，分别表示最左边的前缀字符串、分隔符和最右边的后缀字符串。你可以尝试将代码修改为使用 `mkSting("[", ", ", "]")`，并观察修改后代码的输出。

我们把 `mkString` 方法的输出保存到一个变量之中，再调用 `println` 方法打印这个变量。我们本可以在整个 `map` 方法之外再封装 `println` 方法进行打印，不过此处引入新变量能增强代码的可读性。

# 1.4 并发

Scala 有许多诱人之处，能够使用 Akka API 通过直观的 actor 模式构建健壮的并发应用便是其中之一（请参考 http://akka.io）。

下面的示例有些激进，不过却能让我们体会到 Scala 的强大和优雅。将 Scala 与一套直观的并发 API 相结合，便能以如此简洁优雅的方式实现并发软件。你之前研究 Scala 的一个原因可能是寻求更好的并发之道，以便更好地利用多核 CPU 和集群中的服务器来实现并发。使用 actor 并发模型便是其中的一种方法。

在 actor 并发模型中，actor 是独立的软件实体，它们之间并不共享任何可变状态信息。actor 之间无须共享信息，通过交换消息的方式便可进行通信。通过消除同步访问那些共享可变状态，编写健壮的并发应用程序变得非常简单。尽管这些 actor 也许需要修改状态，但是假如这些可变状态对外不可访问，并且 actor 框架确保 actor 相关代码调用是线程安全的，开发者就无须再费力编写枯燥而又容易出错的*同步原语*（synchronization primitive）了。

在这个简单示例中，我们会将表示几何图形的一组类的实例发送给一个 actor，该 actor 再将这组实例绘制到显示器上。你可以想象这样一个场景：*渲染工厂*（rendering farm）在为动画生成场景。一旦场景渲染完毕，构成场景的几何图形便会被发送给某一 actor 进行展示。

首先，我们将定义 `Shape` 类。

```
// src/main/scala/progscala2/introscala/shapes/Shapes.scala
package progscala2.introscala.shapes

case class Point(x: Double = 0.0, y: Double = 0.0)                  // ❶
```

```
abstract class Shape() {                                           // ❷
  /**
   * draw方法接受一个函数参数。每个图形对象都会将自己的字符格式传给函数f，
   * 由函数f执行绘制工作。
   */
  def draw(f: String => Unit): Unit = f(s"draw: ${this.toString}")    // ❸
}

case class Circle(center: Point, radius: Double) extends Shape       // ❹

case class Rectangle(lowerLeft: Point, height: Double, width: Double) // ❺
      extends Shape

case class Triangle(point1: Point, point2: Point, point3: Point)     // ❻
      extends Shape
```

❶ 此处声明了一个表示二维点的类。

❷ 此处声明了一个表示几何形状的抽象类 。

❸ 此处实现了一个“绘制”形状的 `draw` 方法，该方法中仅输出了一个格式化的字符串。

❹ `Circle` 类由圆心和半径组成。

❺ 位于左下角的点、高度和宽度这三个属性构成了矩形。为了简化问题，我们规定矩形的各条边分别与横坐标或纵坐标平行。

❻ 三角形由三个点所构成。

`Point` 类名后列出的参数列表就是类构造函数参数列表。在 Scala 中，整个类主体便是这个类的构造函数，因此你能在类名之后、类主体之前列出主构造函数的参数。在本示例中，`Point` 类并没有类主体。由于我们在 `Point` 类声明的前面输入了 `case` 关键字，因此每一个构造函数参数都自动转化为 `Point` 实例的某一只读（不可变）字段。也就是说，假如要实例化一个名为 `point` 的 `Point` 实例，你可以使用 `point.x` 和 `point.y` 读取 `point` 的字段，但无法修改它们的值。尝试运行 `point.y = 3.0` 会触发编译错误。

你也可以设置参数默认值。每个参数定义后出现 `= 0.0` 会把 `0.0` 设置为该参数的默认值。因此用户无须明确给出参数值，Scala 便会推导出参数值。不过这些参数值会按照从左到右的顺序进行推导。下面我们运用 SBT 项目去进一步探索参数默认值：

```
$ sbt
...
> compile
Compiling ...
[success] Total time: 15 s, completed ...
> console
[info] Starting scala interpreter...

scala> import progscala2.intro.shapes._
import progscala2.intro.shapes._

scala> val p00 = new Point
p00: intro.shapes.Point = Point(0.0,0.0)
```

```
scala> val p20 = new Point(2.0)
p20: intro.shapes.Point = Point(2.0,0.0)

scala> val p20b = new Point(2.0)
p20b: intro.shapes.Point = Point(2.0,0.0)

scala> val p02 = new Point(y = 2.0)
p02: intro.shapes.Point = Point(0.0,2.0)

scala> p00 == p20
res0: Boolean = false

scala> p20 == p20b
res1: Boolean = true
```

因此，当我们不指定任何参数时，Scala 会使用 `0.0` 作为参数值。当我们只设定了一个参数值时，Scala 会把这个值赋予最左边的参数 `x`，而剩下来的参数则使用默认值。我们还可以通过名字指定参数。对于 `p02` 对象，当我们想使用 `x` 的默认值却为 `y` 赋值时，可以使用 `Point(y = 2.0)` 的语句。

由于 `Point` 类并没有类主体，`case` 关键字的另一个特征便是让编译器自动为我们生成许多方法，其中包括了类似于 Java 语言中 `String`、`equals` 和 `hashCode` 方法。每个点显示的输出信息，如 `Point(2.0,0.0)`，其实是 `toString` 方法的输出。大多数开发者很难正确地实现 `equals` 方法和 `hashCode` 方法，因此自动生成这些方法具有实际的意义。

Scala 调用生成的 `equals` 方法，以判断 `p00 == p20` 和 `p20 == p20b` 是否成立。这与 Java 的做法不同，Java 通过比较引用是否相同来判断 == 是否成立。在 Java 中如果希望执行一次逻辑比较，你需要明确地调用 `equals` 方法。

现在我们要谈论 `case` 类的最后一个特性，编译器同时会生成一个伴生对象（companion object），伴生对象是一个与 `case` 类同名的单例对象（本示例中，`Point` 对象便是一个伴生对象）。

> 你可以自己定义伴生对象。任何时候只要对象名和类名相同并且定义在同一个文件中，这些对象就能称作伴生对象。

随后可以看到，我们可以在伴生对象中添加方法。不过伴生对象中已经自动添加了不少方法，`apply` 方法便是其中之一。该方法接受的参数列表与构造函数接受的参数列表一致。

任何时候只要你在输入对象后紧接着输入一个参数列表，Scala 就会查找并调用该对象的 `apply` 方法，这也意味着下面两行代码是等价的。

```
val p1 = Point.apply(1.0, 2.0)
val p2 = Point(1.0, 2.0)
```

如果对象中未定义 `apply` 方法，系统将抛出编译错误。与此同时，输入参数必须与预期输入相符。

Point.apply 方法实际上是构建 Point 对象的工厂方法，它的行为很简单；调用该方法就好像是不通过 new 关键字调用 Point 的构造函数一样。伴生对象其实与下列代码生成的对象无异。

```
object Point {
  def apply(x: Double = 0.0, y: Double = 0.0) = new Point(x, y)
  ...
}
```

不过，伴生对象 apply 方法也可以用于决定相对复杂的类继承结构。父类对象需判断参数列表与哪个字类型最为吻合，并依此选择实例化的子类型。比方说，某一数据类型必须分别为元素数量少的情况和元素数量多的情况各提供一个不同的最佳实现，此时选用工厂方法可以屏蔽这一逻辑，为用户提供统一的接口。

紧挨着对象名输入参数列表时，Scala 会查找并调用匹配该参数列表的 apply 方法。换句话说，Scala 会猜想该对象定义了 apply 方法。从句法角度上说，任何包含了 apply 方法的对象的行为都很像函数。

在伴生对象中安置 apply 方法是 Scala 为相关类定义工厂方法的一个便利写法。在类中定义而不是在对象中定义的 apply 方法适用于该类的实例。例如，调用 Seq.apply(index:Int) 方法将获得序列中指定位置的元素（从 0 开始计数）。

Shape 是一个抽象类。在 Java 中我们无法实例化一个抽象类，即使该抽象类中没有抽象成员。该类定义了 Shape.draw 方法，不过我们只希望能够实例化具体的形状：圆形、矩阵或三角形。

请注意传给 draw 方法的参数，该参数是一个类型为 String => Unit 的函数。也就是说，函数 f 接受字符串参数输入并返回 Unit 类型。Unit 是一个实际存在的类型，它的表现却与 Java 中的 void 类型相似。在函数式编程中，大家将 void 类型称为 Unit 类型 .

具体做法是 draw 方法的调用者将传入一个函数，该函数会接受表示具体形状的字符串，并执行实际的绘图工作。

假如某函数返回 Unit 对象，那么该函数肯定是有副作用的。Unit 对象没有任何作用，因此该函数只能对某些状态产生副作用。副作用可能会造成全局范围的影响，比如执行一次输入或输出操作（I/O），也可能只会影响某些局部对象。

通常在函数式编程中，人们更青睐于那些没有任何副作用的纯函数，这些纯函数的返回值便是它们的工作成果。纯函数容易阐述、易于测试，也很方便重用，而副作用往往是错误之源。不过最起码现实中的程序离不开 I/O。

Shape.draw 阐明了这样一个观点：与 Strings、Ints、Points 和其他对象无异，函数也是第一等级的值。和其他值一样，我们可以将函数赋给变量，将函数作为参数传递给其他函数，就好像 draw 方法一样。函数还能作为其他函数的返回值。我们将利用函数这一特性构

建可组合并且灵活的软件。

假如某函数接受其他函数参数并返回函数，我们称之为高阶函数（higher-order function，HOF）。

我们可以认为 `draw` 方法定义了一个所有形状类都必须支持的协议，而用户可以自定义这个协议的实现。各个形状类可以通过 `toString` 方法决定如何将状态信息序列化为字符串。`draw` 方法会调用 `f` 函数，而 `f` 函数通过 Scala 2.10 引入的新特性插值字符串（interpolated string）构建了最终的字符串。



如果你忘了在“插值字符串”前输入 `s` 字符，`draw: ${this.toString}` 将原封不动地返回给你。也就是说，字符串不会被窜改。

`Circle`、`Rectangle` 和 `Triangle` 类都是 `Shape` 类的具体子类。这些类并没有类主体，这是因为 `case` 关键字为它们定义好了所有必须的方法，如 `Shape.draw` 所需要的 `toString` 方法。

为了简化问题，我们规定矩形的各条边平行于 *x* 或 *y* 轴。因此，我们使用一个点（左侧最低点即可）、矩形的高度和宽度便能描述矩阵。而 `Triangle` 类（三角形）的构造函数则接受三个 `Pointer` 对象参数。

在简化后的程序中，传递给 `draw` 方法的 `f` 函数只会在控制台中输出一条字符串，不过你也许有机会构建一个真实的图形程序，该程序将使用 `f` 函数将图形绘制到显示器上。

既然已经定义好了形状类型，我们便可以回到 actor 上。其中，Typesafe（http://typesafe.com）贡献的 Akka 类库（http://akka.io）会被使用到。项目文件 build.sbt 中已经将该类库设定为项目依赖项。

下面列出 `ShapesDrawingActor` 类的实现代码：

```
// src/main/scala/progscala2/introscala/shapes/ShapesDrawingActor.scala
package progscala2.introscala.shapes

object Messages {                                                // ❶
  object Exit                                                    // ❷
  object Finished
  case class Response(message: String)                           // ❸
}

import akka.actor.Actor                                          // ❹

class ShapesDrawingActor extends Actor {                         // ❺
  import Messages._                                              // ❻

  def receive = {                                                // ❼
    case s: Shape =>
      s.draw(str => println(s"ShapesDrawingActor: $str"))
```

```
        sender ! Response(s"ShapesDrawingActor: $s drawn")
      case Exit =>
        println(s"ShapesDrawingActor: exiting...")
        sender ! Finished
      case unexpected =>  // default. Equivalent to "unexpected: Any"
        val response = Response(s"ERROR: Unknown message: $unexpected")
        println(s"ShapesDrawingActor: $response")
        sender ! response
    }
}
```

❶ 此处声明了对象 `Messages`，该对象定义了大多数 actor 之间进行通信的消息。这些消息就好像信号量一样，触发了彼此的行为。将这些消息封装在一个对象中是一个常见的封装方式。

❷ `Exit` 和 `Finished` 对象中不包含任何状态，它们起到了标志的作用。

❸ 当接收到发送者发送的消息后，*样板类*（case class）`Response` 会随意构造字符串消息，并将消息返回给发送者。

❹ 导入 `akka.actor.Actor` 类型（http://doc.akka.io/api/akka/current/#akka.actor.Actor）。`Actor` 类型是一个抽象基类，我们将继承该类定义 actor。

❺ 此处定义了一个 actor 类，用于绘制图形。

❻ 此处导入了 `Messages` 对象中定义的三个消息。Scala 支持*嵌套导入*（nesting import），嵌套导入会限定这些值的作用域。

❼ 此处实现了抽象方法 `Actor.receive`。该方法是 `Actor` 的子类必须实现的方法，定义了如何处理接收到的消息。

包括 Akka 在内的大多数 actor 系统中，每一个 actor 都会有一个关联邮箱（mailbox）。关联邮箱中存储着大量消息，而这些消息只有经过 actor 处理后才会被提取。Akka 确保了消息处理的顺序与接收顺序相同，而对于那些正在被处理的消息，Akka 保证不会有其他线程抢占该消息。因此，使用 Akka 编写的消息处理代码天生具有线程安全的特性。

需要注意的是，Akka 支持一种奇特的 `receive` 方法实现方式。该实现不接受任何参数，而实现体中也只包含了一组由 `case` 关键字开头的表达式。

```
def receive = {
  case first_pattern =>
    first_pattern_expressions
  case second_pattern =>
    second_pattern_expressions
}
```

偏函数（`PartialFunction`，http://www.scala-lang.org/api/current/#scala.PartialFunction）是一类较为特殊的函数，上述函数体所用的语法就是典型的偏函数语法。偏函数实际类型是 `PartialFunction[Any,Unit]`，这说明偏函数接受单一的 `Any` 类型参数并返回 `Unit` 值。`Any` 是 Scala 类层次级别的根类，因此该函数可以接受任何参数。由于该函数返回 `Unit` 对象，因此函数体一定会产生副作用。由于 actor 系统采用了异步消息机制，它必须依靠副作用。通常情况下由于传递消息后无法返回任何信息，我们的代码块中便会发送一些其他消息，

包括给发送者的返回信息。

偏函数中仅包含了一些 case 子句，这些子句会对传递给函数的消息执行模式匹配。代码中并没有任何表示消息的函数参数，内部实现需要处理这些消息。

当匹配上某一模式时，系统将执行从箭头符（=>）到下一个 case 子句（也有可能是函数结尾处）之间的表达式。由于箭头符和下一个 case 关键字能够无误地标识代码区间，因此无须使用大括号包住表达式。另外，假如 case 关键字后只有一句简短的表达式，可以不用换行，直接将表达式放在箭头后面。

尽管听上去挺复杂，实际上偏函数是一个简单的概念。单参数函数会接受某一类型的输入值并返回相同或不同类型的值。而选用偏函数相当于明确地告诉其他人："我也许无法处理所有你输入给我的值。"除法 x/y 是数学上的一个经典偏函数例子，当分母 y 为 0 时，x/y 的值是不确定的。因此，除法是一个偏函数。

receive 方法会尝试将接收到的各条消息与这三个模式匹配表达式进行匹配，并执行最先被匹配上的表达式。接下来我们对 receive 方法进行分解。

```
def receive = {
  case s: Shape =>                                          // ❶
    ...
  case Exit =>                                              // ❷
    ...
  case unexpected =>                                        // ❸
    ...
}
```

❶ 如果收到的信息是 Shape 的一个实例，那说明该消息匹配了第一条 case 子句。我们也会将 Shape 对象引用赋给变量 s。也就是说，虽然输入消息的类型为 Any，但 s 类型却是 Shape。

❷ 判断消息是否为 Exit 消息体。Exit 消息用于标识已经完成。

❸ 这是一条"默认"子句，可以匹配任何输入。该子句等同于 unexpected: Any 子句，对于那些未能与前两个子句模式匹配的任何输入，该子句都会匹配。而变量 unexpected 会被赋予消息值。

最后一条匹配规则能匹配任何消息，因此该规则必须放到最后一位。假如你尝试将其放置到某些规则之前，你将看到 unreachable code 的错误信息。这是因为这些后续的 case 表达式不可访问。

值得注意的是，由于我们添加了"默认"子句，这个"偏"函数其实变成了"完整的"，这意味着该函数能正确处理任何输入。

下面让我们查看每个匹配点调用的表达式：

```
def receive = {
  case s: Shape =>
    s.draw(str => println(s"ShapesDrawingActor: $str"))     // ❶
    sender ! Response(s"ShapesDrawingActor: $s drawn")      // ❷
  case Exit =>
    println(s"ShapesDrawingActor: exiting...")              // ❸
```

```
      sender ! Finished                                        // ❹
    case unexpected =>
      val response = Response(s"ERROR: Unknown message: $unexpected")  // ❺
      println(s"ShapesDrawingActor: $response")
      sender ! response                                        // ❻
  }
```

❶ 调用了形状 s 的 draw 方法并传入一个匿名函数，该匿名函数了解如何处理 draw 方法生成的字符串。在这段代码中，此匿名函数仅打印了生成的字符串。

❷ 向“发信方”回复了一个消息。

❸ 打印了一条表示正在退出的消息。

❹ 向“发信方”发送了一条结束信息。

❺ 根据错误信息生成 Response 对象，并打印错误信息。

❻ 向“发信方”回复了这条信息。

代码 sender ! Response(s"ShapesDrawingActor: $s drawn") 创建了回复信息，并将该信息发送给了 shape 对象的发送方。Actor.sender 函数返回了 actor 发送消息接收方的对象引用，而 ! 方法则用于发送异步消息。是的，! 是一个方法名，使用 ! 遵循了之前 Erlang 的消息发送规范，值得一提的是，Erlang 是一门推广 actor 模型的语言。

我们也可以在 Scala 允许范围内使用一些语法糖。下面两行代码是等价的：

```
sender ! Response(s"ShapesDrawingActor: $s drawn")
sender.!(Response(s"ShapesDrawingActor: $s drawn"))
```

假如某一方法只接受单一参数，你可以省略掉对象后的点号和参数周边的括号。请注意，第一行代码看起来更清晰，这也是 Scala 支持这种语法的原因。表示法 sender ! Response 被称为中置表示法，这是因为操作符 ! 位于对象和参数中间。

> Scala 的方法名可以是操作符。调用接受单一参数的方法时可以省略对象后的点号和参数周边的括号。不过有时候省略它们会导致解析二义性，这时你需要保留点号或保留括号，有时候两者都需要保留。

在进入最后一个 actor 之前还有最后一个值得注意的地方。使用面向对象编程时，有一条经常被人提及的原则：永远不要在 case 语句上进行类型匹配。这是因为如果继承层次结构发生了变化，case 表达式也会失效。作为替代方案，你应该使用多态函数。这是不是意味着我们之前谈论的模式匹配代码只是一个反模式呢？

回顾一下，我们之前定义的 Shape.draw 方法调用了 Shape 类的 toString 方法，由于 Shape 类的那些子类是 case 类，因此这些子类中实现了 toString 方法。第一个 case 语句中的代码调用了多态的 toString 操作，而我们也没有与 Shape 的某一具体子类进行匹配。这意味着即便修改了 Shape 类层次结构，我们的代码也不会失效。其他的 case 子句所匹配的条件也与类层次无关，即便这些条件真会发生变化，变化也不会频繁。

由此，我们将面向对象编程中的多态与函数式编程中的劳模——模式匹配结合到了一起。

这是 Scala 优雅地集成这两种编程范式的方式之一。

> **模式匹配与子类型多态**
>
> 模式匹配在函数式编程中扮演了重要的角色，而子类型多态（即重写子类型中的方法）在面向对象编程的世界中同样不可或缺。函数式编程中的模式匹配的重要性和复杂度都要远超过大多数命令式语言中对应的 swith/case 语句。我们将在第 4 章深入探讨模式匹配。在此处的示例中，我们开始了解到函数风格的模式匹配和多态调度之间的结合会产生强大的组合效果，而这也是像 Scala 这样的混合范式语言能提供的一大益处。

最后，我将列出运行此示例的 ShapesDrawingDriver 对象的代码：

```
// src/main/scala/progscala2/introscala/shapes/ShapesActorDriver.scala
package progscala2.introscala.shapes
import akka.actor.{Props, Actor, ActorRef, ActorSystem}
import com.typesafe.config.ConfigFactory

// 仅用于本文件的消息：
private object Start                                                // ❶

object ShapesDrawingDriver {                                        // ❷
  def main(args: Array[String]) {                                   // ❸
    val system = ActorSystem("DrawingActorSystem", ConfigFactory.load())
    val drawer = system.actorOf(
      Props(new ShapesDrawingActor), "drawingActor")
    val driver = system.actorOf(
       Props(new ShapesDrawingDriver(drawer)), "drawingService")
    driver ! Start                                                  // ❹
  }
}

class ShapesDrawingDriver(drawerActor: ActorRef) extends Actor {     // ❺
import Messages._

  def receive = {
    case Start =>                                                   // ❻
      drawerActor ! Circle(Point(0.0,0.0), 1.0)
      drawerActor ! Rectangle(Point(0.0,0.0), 2, 5)
      drawerActor ! 3.14159
      drawerActor ! Triangle(Point(0.0,0.0), Point(2.0,0.0), Point(1.0,2.0))
      drawerActor ! Exit
    case Finished =>                                                // ❼
      println(s"ShapesDrawingDriver: cleaning up...")
      context.system.shutdown()
    case response: Response =>                                      // ❽
      println("ShapesDrawingDriver: Response = " + response)
    case unexpected =>                                              // ❾
      println("ShapesDrawingDriver: ERROR: Received an unexpected message = "
        + unexpected)
  }
}
```

❶ 定义仅用于本文件的消息（私有消息），该消息用于启动。使用一个特殊的开始消息是一个普遍的做法。

❷ 定义“驱动”actor。

❸ 定义了用于驱动应用的主方法。主方法先后构建了一个 `akka.actor.ActorSystem` 对象（http://doc.akka.io/api/akka/current/#akka.actor.ActorSystem） 和 两 个 actor 对 象：我们之前讨论过的 `ShapesDrawingActor` 对象和即将讲解的 `ShapesDrawingDriver` 对象。我们暂时先不讨论设置 Akka 的方法，在 17.3 节将详细讲述。现在只需要知道我们把 `ShapesDrawingActor` 对象传递给了 `ShapesDrawingDriver` 即可，事实上我们向 `ShapesDrawingDriver` 对象传递的对象属于 `akka.actor.ActorRef` 类型（http://doc.akka.io/api/akka/current/#akka.actor.ActorRef，actor 的引用类型，指向实际的 actor 实例）。

❹ 向驱动对象发送 `Start` 命令，启动应用！

❺ 定义了 actor 类：`ShapesDrawingDriver`。

❻ 当 `receive` 方法接收到 `Start` 消息时，它将向 `ShapesDrawingActor` 发送五个异步消息：包含了三个形状类对象，`Pi` 值（将被视为错误信息）和 `Exit` 消息。从这能看出，这是一个生命周期很短的 actor 系统！

❼ 假如 `ShapesDrawingDriver` 发送 `Exit` 消息后接收到了返回的 `Finished` 消息（请回忆一下 `ShapesDrawingActor` 类处理 `Exit` 消息的逻辑），那么我们将访问 `Actor` 类提供的 `context` 字段来关闭 actor 系统。

❽ 简单地打印出其他错误的回复信息。

❾ 与之前所见的默认子句一样，该子句用于处理预料之外的消息。

让我们尝试运行该程序！在 `sbt` 提示符后输入 **`run`**，`sbt` 将按需编译代码并列出所有定义了 `main` 方法的代码示例程序：

```
> run
[info] Compiling ...

Multiple main classes detected, select one to run:

 [1] progscala2.introscala.shapes.ShapesDrawingDriver
 ...

Enter number:
```

输入数字 **`1`**，之后我们便能看到下列输出 ( 为了方便显示，已对输出内容进行排版 )：

```
...
Enter number: 1

[info] Running progscala2.introscala.shapes.ShapesDrawingDriver
ShapesDrawingActor: draw: Circle(Point(0.0,0.0),1.0)
ShapesDrawingActor: draw: Rectangle(Point(0.0,0.0),2.0,5.0)
ShapesDrawingActor: Response(ERROR: Unknown message: 3.14159)
ShapesDrawingActor: draw: Triangle(
  Point(0.0,0.0),Point(2.0,0.0),Point(1.0,2.0))
ShapesDrawingActor: exiting...
```

```
ShapesDrawingDriver: Response = Response(
  ShapesDrawingActor: Circle(Point(0.0,0.0),1.0) drawn)
ShapesDrawingDriver: Response = Response(
  ShapesDrawingActor: Rectangle(Point(0.0,0.0),2.0,5.0) drawn)
ShapesDrawingDriver: Response = Response(ERROR: Unknown message: 3.14159)
ShapesDrawingDriver: Response = Response(
  ShapesDrawingActor: Triangle(
    Point(0.0,0.0),Point(2.0,0.0),Point(1.0,2.0)) drawn)
ShapesDrawingDriver: cleaning up...
[success] Total time: 10 s, completed Aug 2, 2014 7:45:07 PM
>
```

由于所有的消息都是以异步的方式发送的，你可以看到驱动 actor 和绘图 actor 的消息交织在一起。不过处理消息的顺序与发送消息的顺序相同。运行多次应用程序，你会发现每次输出都会不同。

到现在为止，我们已经尝试了基于 actor 的并发编程，同时也掌握了一些很有威力的 Scala 特性。

## 1.5 本章回顾与下一章提要

我们首先介绍了 Scala，之后分析了一些重要的 Scala 代码，其中包含一些 Akka 的 actor 并发库相关代码。

你在学习 Scala 的过程中，也可以访问 http://scala-lang.org 网站获取其他一些有用的资源。在该网站上，能找到一些指向 Scala 类库、教程以及一些描述这门语言特性相关文章的链接。

Typesafe 是一家为 Scala 以及包括 Akka（http://akka.io）、Play（http://www.playframework.com）在内的许多基于 JVM 的开发工具和框架提供支持的商业公司。在该公司的网站上（http://typesafe.com）也能找到一些有用的资源。尤其是 Typesafe Activator 工具（http://typesafe.com/activator），该工具会根据不同类型的 Scala 或 Java 应用程序模版，执行分析、下载和构建工作。Typesafe 公司还提供了订购支持、咨询及培训服务。

在后续的部分，我们将继续介绍 Scala 的特性，着重介绍如何使用 Scala 简洁有效地完成工作。

# 第 2 章

# 更简洁，更强大

在第 1 章的结尾我们介绍了一个关于 Akka actor 的应用，这个例子可能稍显复杂。本章我们将继续探究 Scala 的特性，重点关注它如何为我们提供简洁且灵活的语法代码。我们将探讨如何组织文件与包，如何导入其他类型、变量、方法声明，以及一些非常有用的数据类型和各种约定俗成的语法习惯。

## 2.1 分号

分号是表达式之间的分隔符，可以推断得出。当一行结束时，Scala 就认为表达式结束了，除非它可以推断出该表达式尚未结束，应该延续到下一行，如下面这个例子：

```
// src/main/scala/progscala2/typelessdomore/semicolon-example.sc

// 末尾的等号表明下一行还有未结束的代码。
def equalsign(s: String) =
  println("equalsign: " + s)

// 末尾的花括号表明下一行还有未结束的代码。
def equalsign2(s: String) = {
  println("equalsign2: " + s)
}

// 末尾的逗号、句号和操作符都可以表明,下一行还有未结束的代码。
def commas(s1: String,
           s2: String) = Console.
  println("comma: " + s1 +
          ", " + s2)
```

与编译器相比，REPL 更容易将每行视为单独的表达式。因此，在 REPL 中输入跨越多行

的表达式时，最安全的做法是每行（除最后一行外）都以上述脚本中出现过的符号结尾。

反过来，你可以将多个表达式放在同一行中，表达式之间用分号隔开。

如果你需要将多行代码解释为同一表达式，却被系统视为多个表达式，可以使用 REPL 的 `:paste` 模式。输入 `:paste`，然后输入你的代码，最后用 Ctrl-D 结束。

## 2.2 变量声明

在声明变量时，Scala 允许你决定该变量是不可变（只读）的，还是可变的（读写）。如下所示，不可变的“变量”用 `val` 关键字声明：

```
val array: Array[String] = new Array(5)
```

Scala 的大部分变量事实上是指向堆内存对象的引用，这一点与 Java 一致。所以，以上代码中的 `array` 也是一个引用，它不能指向其他 `Array`，但所指向的 `Array` 中的元素是可变的，如下所示：

```
scala> val array: Array[String] = new Array(5)
array: Array[String] = Array(null, null, null, null, null)

scala> array = new Array(2)
<console>:8: error: reassignment to val
       array = new Array(2)

scala> array(0) = "Hello"

scala> array
res1: Array[String] = Array(Hello, null, null, null, null)
```

一个 `val` 变量在声明时必须被初始化。

类似地，一个可变变量用关键字 `var` 来声明。尽管由于该变量是可变变量，声明后可以再次对其赋值，也必须在声明的同时立即初始化：

```
scala> var stockPrice: Double = 100.0
stockPrice: Double = 100.0

scala> stockPrice = 200.0
stockPrice: Double = 200.0
```

这里要区分一下：这一次我们修改了 `stockPrice` 本身，然而，`stockPrice` 所引用的“对象”没有被修改，因为在 Scala 中 `Double` 类型是不可变的。

在 Java 中，所谓的原生类型，即 `char`、`byte`、`short`、`int`、`long`、`float`、`double` 和 `boolean`，与其他引用类型有着本质的不同。这些类型确实既不是对象，也没有引用，是“原始”值。Scala 尽力使其面向对象特性更加一致，因此这些类型在 Scala 中是包含有方法的对象，就像引用类型一样（参见 8.2 节）。然而，Scala 编译时将这些类型尽可能地转为原生类型，使你可以得到原生类型的运行效率（在 12.4 节中我们将深入讨论这一点）。

用 val 和 var 声明变量时必须初始化这一规则，但存在少数例外情况。例如，这两个关键字均可以用在构造函数的参数中，这时候变量是该类的一个属性，因此显然不必在声明时进行初始化。此时如果用 val 声明，该属性是不可变的；如果用 var 声明，则该属性是可变的。

考虑如下 REPL 会话，在这里我们定义了 Person 类，其中包含表示姓和名的不可变变量，而年龄则是可变的（因为人的年龄会随时间变大的缘故）：

```
// src/main/scala/progscala2/typelessdomore/person.sc
scala> class Person(val name: String, var age: Int)
defined class Person

scala> val p = new Person("Dean Wampler", 29)
p: Person = Person@165a128d

scala> p.name
res0: String = Dean Wampler     // 显示firstName的值

scala> p.age
res2: Int = 29                  // 显示age的值

scala> p.name = "Buck Trends"
<console>:9: error: reassignment to val   // 这是不允许的!
       p.name = "Buck Trends"
              ^

scala> p.age = 30
p.age: Int = 30                 // 允许!
```

> var 和 val 关键字只标识引用本身是否可以指向另一个不同的对象，它们并未表明其所引用的对象是否可变。

为了减少可变性引起的 bug，应该尽可能地使用不可变变量。

例如，在散列映射中，可变对象是非常危险的。如果对象发生改变，hashCode 方法的输出就会发生变化，在散列映射表中原来的位置就无法找到对应的值了。

更为常见的是，当你正在使用的对象被其他人修改时，将引起对象产生不可预见的行为。借用量子力学的名词，这是一种“幽灵般的超距作用”，本地的所有操作都无法解释这种不可预见的行为，因为这是由其他某处的操作引起的。

这在多线程程序中是最致命的 bug。在多线程程序中，对共享的可变状态进行读写之前要使用同步操作，但实践中往往很难实现正确的同步。

这个时候，如果你使用的是不可变的值，就可以减少这类问题。

## 2.3 Range

我们接下来将讨论方法的声明，但其中的示例需要用到 Range 的概念（http://www.scala-lang.org/api/current/scala/collection/immutable/Range.html），因此我们先来讨论 Range。

有时我们需要一个数字序列，从某个起点到某个终点。而 Range 能满足这个需要。以下实例将展示如何创建 Range，支持 Range 的类型包括 Int、Long、Float、Double、Char、BigInt（支持任意大小的整数，http://www.scala-lang.org/api/current/scala/math/BigInt.html）和 BigDecimal（支持任意大小的浮点数，http://www.scala-lang.org/api/current/scala/math/BigDecimal.html）。

你创建的 Range 可以包含区间上限，也可以不包含区间上限；步长默认为 1，也可以指定一个非 1 的步长：

```
scala> 1 to 10                // Int类型的Range,包括区间上限,步长为1(从1到10)
res0: scala.collection.immutable.Range.Inclusive =
  Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> 1 until 10             // Int类型的Range,不包括区间上限,步长为1(从1到9)
res1: scala.collection.immutable.Range = Range(1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> 1 to 10 by 3           // Int类型的Range,包括区间上限,步长为3
res2: scala.collection.immutable.Range = Range(1, 4, 7, 10)

scala> 10 to 1 by -3          // Int类型的递减Range,包括区间下限,步长为-3
res2: scala.collection.immutable.Range = Range(10, 7, 4, 1)

scala> 1L to 10L by 3         // Long类型
res3: scala.collection.immutable.NumericRange[Long] = NumericRange(1, 4, 7, 10)

scala> 1.1f to 10.3f by 3.1f  // Float类型的Range,步长可以不等于1
res4: scala.collection.immutable.NumericRange[Float] =
  NumericRange(1.1, 4.2, 7.2999997)

scala> 1.1f to 10.3f by 0.5f  // Float类型的Range,步长可以小于1
res5: scala.collection.immutable.NumericRange[Float] =
  NumericRange(1.1, 1.6, 2.1, 2.6, 3.1, 3.6, 4.1, 4.6, 5.1, 5.6, 6.1, 6.6,
    7.1, 7.6, 8.1, 8.6, 9.1, 9.6, 10.1)

scala> 1.1 to 10.3 by 3.1     // Double类型
res6: scala.collection.immutable.NumericRange[Double] =
  NumericRange(1.1, 4.2, 7.300000000000001)

scala> 'a' to 'g' by 3        // Char类型
res7: scala.collection.immutable.NumericRange[Char] = NumericRange(a, d, g)

scala> BigInt(1) to BigInt(10) by 3
res8: scala.collection.immutable.NumericRange[BigInt] =
  NumericRange(1, 4, 7, 10)

scala> BigDecimal(1.1) to BigDecimal(10.3) by 3.1
res9: scala.collection.immutable.NumericRange.Inclusive[scala.math.BigDecimal]
  = NumericRange(1.1, 4.2, 7.3)
```

部分输出根据页面大小进行了重新排版。

## 2.4 偏函数

我们来讨论偏函数（PartialFunction，http://bit.ly/1yMpzEP）的性质。偏函数之所以“偏”，原因在于它们并不处理所有可能的输入，而只处理那些能与至少一个 case 语句匹配的输入。

在偏函数中只能使用 case 语句，而整个函数必须用花括号包围。这与普通的函数字面量不同，普通函数字面量可以用花括号，也可以用圆括号包围。

如果偏函数被调用，而函数的输入却与所有语句都不匹配，系统就会抛出一个 MatchError（http://www.scala-lang.org/api/current/#scala.MatchError）运行时错误。

我们可以用 isDefineAt 方法测试特定输入是否与偏函数匹配，这样偏函数就可以避免抛出 MatchError 错误了。

偏函数可以如此“链式”连接：pf1 orElse pf2 orElse pf3…。如果 pf1 不匹配，就会尝试 pf2，接着是 pf3，以此类推。如果以上偏函数都不匹配，才会抛出 MatchError。

以下实例可以展示上述规则：

```
// src/main/scala/progscala2/typelessdomore/partial-functions.sc

val pf1: PartialFunction[Any,String] = { case s:String => "YES" }    // ❶
val pf2: PartialFunction[Any,String] = { case d:Double => "YES" }    // ❷

val pf = pf1 orElse pf2                                              // ❸

def tryPF(x: Any, f: PartialFunction[Any,String]): String =          // ❹
  try { f(x).toString } catch { case _: MatchError => "ERROR!" }

def d(x: Any, f: PartialFunction[Any,String]) =                      // ❺
  f.isDefinedAt(x).toString

println("      |   pf1 - String  |   pf2 - Double  |    pf - All")   // ❻
println("x     | def?  |  pf1(x) | def?  |  pf2(x) | def?  |  pf(x)")
println("++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++")
List("str", 3.14, 10) foreach { x =>
  printf("%-5s | %-5s | %-6s  | %-5s | %-6s  | %-5s | %-6s\n", x.toString,
    d(x,pf1), tryPF(x,pf1), d(x,pf2), tryPF(x,pf2), d(x,pf), tryPF(x,pf))
}
```

❶ 只匹配字符串的偏函数。

❷ 只匹配 Double 数字的偏函数。

❸ 将这两个函数结合，得到一个新的偏函数：既能匹配字符串，又能匹配 Double 数字。

❹ 辅助函数：用于 try 一个偏函数，然后将可能产生的 MatchError 异常捕捉到。无论是否捕获异常，函数均返回一个字符串。

❺ 辅助函数：使用了 isDefineAt，返回值为字符串。

❻ 使用了多个偏函数的链式组合，并将结果以表格的形式打印出来。

其他代码对这 3 个偏函数输入不同的值，先调用 isDefineAt（结果显示在输出表中的 def? 这一列），然后再尝试调用偏函数本身。输出为：

```
      |   pf1 - String  |   pf2 - Double  |    pf - All
x     | def?  |  pf1(x) | def?  |  pf2(x) | def?  |  pf(x)
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
str   | true  | YES     | false | ERROR!  | true  | YES
3.14  | false | ERROR!  | true  | YES     | true  | YES
10    | false | ERROR!  | false | ERROR!  | false | ERROR!
```

未输入字符串时，pf1 将会失败；未输入 Double 数字时，pf2 会失败；如果给出整数，这两个函数均失败。组合后的函数 pf 对于字符串或者 Double 数字的输入均成功，但输入整数时仍将失败。

# 2.5 方法声明

下面我们来探讨方法的声明。本节我们会用到前文使用的 Shape 类的继承树并加以修改（为了简单处理，我们去掉了 Triangle 类）。

## 2.5.1 方法默认值和命名参数列表

以下是修改后的 Point case 类：

```
// src/main/scala/progscala2/typelessdomore/shapes/Shapes.scala
package progscala2.typelessdomore.shapes

case class Point(x: Double = 0.0, y: Double = 0.0) {              // ❶

  def shift(deltax: Double = 0.0, deltay: Double = 0.0) =         // ❷
    copy (x + deltax, y + deltay)
}
```

❶ 如同前文，定义 Point 类，并提供默认的初始化值。

❷ 新的 shift 方法，用于从现有的 Point 对象中对“点”进行平移，从而创建一个新的 Point 对象。它使用了 copy 方法，copy 方法也是 case 类自动创建的。

copy 方法允许你在创建 case 类的新实例时，只给出与原对象不同部分的参数，这一点对于大一些的 case 类非常有用：

```
scala> val p1 = new Point(x = 3.3, y = 4.4)    // 显式使用命名参数列表。
p1: Point = Point(3.3,4.4)

scala> val p2 = p1.copy(y = 6.6)  // 指定新的y值,创建新实例。
p2: Point = Point(3.3,6.6)
```

命名参数列表让客户端代码更具可读性。当参数列表很长，且有若干参数是同一类型时，bug 容易避免，因为在这种情况下很容易搞错参数传入的顺序。当然，更好的做法是一开始就避免出现过长的参数列表。

### 2.5.2 方法具有多个参数列表

接下来，我们对 Shape 类进行修改，特别是其中的 draw 方法：

```
abstract class Shape() {
  /**
   * draw 带两个参数列表,其中一个参数列表带着一个表示绘制偏移量的参数
   * 另一个参数列表是我们之前用过的函数参数。
   */
  def draw(offset: Point = Point(0.0, 0.0))(f: String => Unit): Unit =
    f(s"draw(offset = $offset), ${this.toString}")
}

case class Circle(center: Point, radius: Double) extends Shape

case class Rectangle(lowerLeft: Point, height: Double, width: Double)
  extends Shape
```

没错，这里的 draw 方法有两个参数列表，每个参数列表都有一个参数，而不是拥有一个具有两个参数的参数列表。第一个参数列表允许你指定 Point 对象的偏移量，供绘制使用。默认值 Point(0.0, 0.0)，表示没有偏移。第二个参数列表与之前的 draw 函数相同，其中的参数是用来绘制所用的函数的。

你可以任意指定参数列表的个数，但实际上很少有人使用两个以上的参数列表。

那么，为什么要允许多个参数列表呢？当最后一个参数列表只包含一个表示函数的参数时，多个参数列表的形式拥有整齐的块结构语法。以下是我们调用新的 draw 方法的表达方式：

```
s.draw(Point(1.0, 2.0))(str => println(s"ShapesDrawingActor: $str"))
```

Scala 允许我们把参数列表两边的圆括号替换为花括号，因此，这一行代码还可以写为：

```
s.draw(Point(1.0, 2.0)){str => println(s"ShapesDrawingActor: $str")}
```

如果函数字面量不能在一行内完成，我们可以重写为以下方式：

```
s.draw(Point(1.0, 2.0)) { str =>
  println(s"ShapesDrawingActor: $str")
}
```

或写为等价的形式：

```
s.draw(Point(1.0, 2.0)) {
  str => println(s"ShapesDrawingActor: $str")
}
```

这一写法很像我们之前常用来写 if 和 for 表达式或方法体的代码块。只不过，在这里的 {…} 块所表示的函数是我们要传递给 draw 方法的参数。

当函数字面量很长时，这种用 {…} 代替 (…) 的“语法糖”使得代码看起来美观多了。此时的代码更像我们所熟悉和喜爱的块结构语法。

如果我们使用缺省的偏移量，第一个圆括号就不能省略：

```
s.draw() {
  str => println(s"ShapesDrawingActor: $str")
}
```

如同 Java 方法一样，draw 方法也可以只使用一个带两个参数值的参数列表。如果那样，客户端代码就会像这样写：

```
s.draw(Point(1.0, 2.0),
  str => println(s"ShapesDrawingActor: $str")
)
```

这份代码并没那么清晰和优雅。使用默认值开启 offset 也没那么便捷，因此我们不得不对参数进行命名：

```
s.draw(f = str => println(s"ShapesDrawingActor: $str"))
```

第二个优势是在之后的参数列表中进行类型推断。如以下例子：

```
scala> def m1[A](a: A, f: A => String) = f(a)
m1: [A](a: A, f: A => String)String

scala> def m2[A](a: A)(f: A => String) = f(a)
m2: [A](a: A)(f: A => String)String

scala> m1(100, i => s"$i + $i")
<console>:12: error: missing parameter type
              m1(100, i => s"$i + $i")
                      ^

scala> m2(100)(i => s"$i + $i")
res0: String = 100 + 100
```

函数 m1 和函数 m2 看起来几乎一模一样，但我们需要注意用相同的参数调用它们时 m1 和 m2 的表现。我们传入 Int 和一个函数 Int => String，对于 m1，Scala 无法推断该函数的参数 i，m2 则可以。

使用多个参数列表的第三个优势是，我们可以用最后一个参数列表来推断隐含参数。隐含参数是用 implicit 关键字声明的参数。当相应方法被调用时，我们可以显式指定这个参数，或者也可以不指定，这时编译器会在当前作用域中找到一个合适的值作为参数。隐含参数可以代替参数默认值，而且更加灵活。我们这就来研究一个 Scala 库中使用隐含参数的例子 Future。

### 2.5.3 Future简介

scala.concurrent.Future（http://www.scala-lang.org/api/current/scala/concurrent/Future.html）是 Scala 提供的一个并发工具，其中的 API 使用隐含参数来减少冗余代码。Akka 使用了 Future，但如果你并不需要 actor 的所有功能，也可以单独使用 Akka 中的 Future 部分。

当你将任务封装在 Future 中执行时，该任务的执行是异步的。Future API 提供了多种机制去获取执行结果，如提供回调函数。当结果就绪时，回调函数将被调用。我们在这里就使用回调函数作为例子，关于其他 API 的讨论将推迟到第 17 章。

以下示例并发发出 5 个任务，并在任务结束时处理任务返回的结果：

```
// src/main/scala/progscala2/typelessdomore/futures.sc
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def sleep(millis: Long) = {
  Thread.sleep(millis)
}
// 繁忙的处理工作;)
def doWork(index: Int) = {
  sleep((math.random * 1000).toLong)
  index
}

(1 to 5) foreach { index =>
  val future = Future {
    doWork(index)
  }
  future onSuccess {
    case answer: Int => println(s"Success! returned: $answer")
  }
  future onFailure {
    case th: Throwable => println(s"FAILURE! returned: $th")
  }
}

sleep(1000)  // 等待足够长的时间,以确保工作线程结束。
println("Finito!")
```

在 import 语句之后的 `sleep` 函数中，我们调用 `Thread` 的 `sleep` 方法来模拟程序进行繁忙的处理工作，参数为睡眠的时长。`doWork` 方法是对 `sleep` 的简单封装，传入一个 0 到 1000 范围的随机数，表示睡眠的毫秒数。

接下来的部分就有趣了。我们用 `foreach` 对一个从 1 到 5 的 `Range` 进行迭代，调用了 `scala.concurrent.Future.apply`（http://www.scala-lang.org/api/current/scala/concurrent/Future.html），这是单例对象 `Future` 的“工厂”方法。在这个例子中，`Future.apply` 传入了一个匿名函数，表示需要做的任务。我们用花括号而不是圆括号包围传入的匿名函数：

```
val future = Future {
  doWork(index)
}
```

`Future.apply` 返回一个新的 `Future` 对象，然后控制权就交还给循环了，该对象将在另一个线程中执行 `doWork(index)`。接着，我们用 `onSuccess` 注册一个回调函数，当 `future` 成功执行完毕后，该回调会被执行。这个回调函数是一个偏函数。

类似地，我们用 `onFailure` 注册了一个回调函数来处理错误。回调函数将错误封装在一个 `Throwable`（http://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html）的对象中。

最后，我们调用了 `sleep`，以等待所有的任务执行完毕。

可能的运行结果如下所示。在本例中结果将以随机的顺序出现，这一点你也许有预期到：

```
$ scala src/main/scala/progscala2/typelessdomore/futures.sc
Success! returned: 1
Success! returned: 3
Success! returned: 4
Success! returned: 5
Success! returned: 2
Finito!
```

好了，上面说的这些与隐含参数有什么关系呢？从第二条 import 语句中可以发现答案：

```
import scala.concurrent.ExecutionContext.Implicits.global
```

我之前说过，每个 `future` 运行在各自独立的线程中，但这个说法不够严谨。事实上，Future API 允许我们通过 `ExecutionContext` 来配置并发操作的执行。上述 import 语句导入了默认的 `ExecutionContext`，使用 Java 的 `ForkJoinPool`（http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html）设置来管理 Java 线程池。在示例中我们调用了 3 个方法，其中这些方法的第二个参数列表具有隐含的 `ExecutionContext` 参数。由于我们没有明显地指定第二个参数列表，系统使用了默认的 `ExecutionContext`。

`Apply` 方法就是上述 3 个方法之一，以下是 `apply` 在 `Future.apply` 的 Scaladoc（http://www.scala-lang.org/api/current/#scala.concurrent.Future$）中的声明：

```
apply[T](body: => T)(implicit executor: ExecutionContext): Future[T]
```

注意第二个参数列表中有 `implicit` 关键字。

以下是其他两个方法在 `Fure.onSuccess` 和 `Future.onFailure` 的 Scaladoc 中的声明：

```
def onSuccess[U](func: (Try[T]) => U)(
  implicit executor: ExecutionContext): Unit
def onFailure[U](callback: PartialFunction[Throwable, U])(
  implicit executor: ExecutionContext): Unit
```

`Try` 结构是一个处理 `try {…} catch {…}` 语句的工具，我们以后再讨论它。

那么，如何将某个值声明为 implicit 呢？导入的 `scala.concurrent.ExecutionContext.Implicits.global` 是在 `Future` 中常用的默认 `ExecutionContext`。它使用 `implicit` 关键字声明，因此如果调用时未显式给出 `ExecutionContext` 参数，编译器就会使用这个默认值，本例就是这种情况。只有由 `implicit` 关键字声明的，在当前作用域可见的对象才能用作隐含值；只有被声明为 implicit 的函数参数才允许调用时不给出实参，而采用隐含的值。

以下就是 Scala 源代码中的 `scala.concurrent.ExecutionContext` 对 `Implicits.global` 的声明。这段代码演示了如何用 `implicit` 关键字声明一个隐含的值（这里只给出了代码片段，省略了具体细节）：

```
object Implicits {
  implicit val global: ExecutionContextExecutor =
    impl.ExecutionContextImpl.fromExecutor(null: Executor)
}
```

在后续的示例中我们会创建自己的隐含变量。

### 2.5.4 嵌套方法的定义与递归

方法的定义还可以嵌套。当你将一个很长的方法重构为几个更小的方法时，如果这些小的辅助方法只在该方法中调用，就可以用嵌套方法。我们将这些辅助函数嵌套定义在原方法中，它们便对其他外层的代码不可见，包括类中的其他方法。

以下代码实现了阶乘的计算，在这个方法中，我们调用了另一个嵌套的方法去完成阶乘的实际计算：

```
// src/main/scala/progscala2/typelessdomore/factorial.sc

def factorial(i: Int): Long = {
  def fact(i: Int, accumulator: Int): Long = {
    if (i <= 1) accumulator
    else fact(i - 1, i * accumulator)
  }

  fact(i, 1)
}

(0 to 5) foreach ( i => println(factorial(i)) )
```

以下为代码运行的输出：

```
1
1
2
6
24
120
```

辅助函数递归地调用它本身，并传入一个 `accumulator` 参数，阶乘的计算结果保存在该参数中。注意，当计数器 `i` 达到 1 时，我们就将阶乘的计算结果返回。（这里我们不考虑负整数参数，负整数的输入是无效的，本函数在 `i <= 1` 时返回 1。）定义好嵌套的方法后，`factorial` 调用该方法，传入参数 `i`，并累计乘法的初始值 1。

> 很容易忘记调用嵌套的函数！如果编译器提示，能找到 `Unit` 但找不到 `Long`，可能就是因为你忘记调用嵌套函数了。

是否注意到，我们两次用 `i` 作为参数名？第一次是 `factorial` 方法的参数，第二次是嵌套的 `fact` 方法的参数。在 `fact` 方法中使用的 `i` 参数“屏蔽”了外部 `factorial` 方法的 `i` 参数。这样做是允许的，因为我们在 `fact` 方法中并不需要外部的 `i`，我们只在 `factorial` 结尾调用 `fact` 的时候才需要它。

类似方法中声明的局部变量，嵌套的方法也只在声明它的方法中可见。

观察这两个方法的返回值。因为阶乘的计算结果增长非常快，我们选择使用 `Long` 类型，而不使用 Scala 自动推断的 `Int` 类型。如果使用 `Int` 类型，`factorial` 就不需要上述的类型注

释了。然而，我们必须要为 fact 声明返回类型。因为这是一个递归方法，Scala 采用的是局部作用域类型推断，无法推断出递归函数的返回类型。

对递归函数你也许会感到一丝不安。我们是否在冒风险？ JVM 和许多其他语言环境并不对尾递归做优化，否则尾递归会将递归转为循环，可以避免栈溢出。（尾递归一词，表示调用递归函数是该函数中最后一个表达式，该表达式的返回值就是所调用的递归函数的返回值。）

递归是函数式编程的特点，也是优雅地实现很多算法的强大工具。所以，Scala 编译器对尾递归做了有限的优化。它会对函数调用自身做优化，但不会优化所谓的 trampoline 的情况，也就是“a 调用 b 调用 a 调用 b”的情形。

你可能仍然想知道自己写的尾递归是否正确，编译器是否对自己的尾递归执行了优化。没有人希望在生产环境中出现栈空间崩溃。幸运的是，如果你加一个 tailrec 关键字（http://www.scala-lang.org/api/current/#scala.annotation.tailrec），编译器会告诉你代码是否正确地实现了尾递归，如以下 factorial 的改良版本：

```
// src/main/scala/progscala2/typelessdomore/factorial-tailrec.sc
import scala.annotation.tailrec

def factorial(i: Int): Long = {
  @tailrec
  def fact(i: Int, accumulator: Int): Long = {
    if (i <= 1) accumulator
    else fact(i - 1, i * accumulator)
  }

  fact(i, 1)
}

(0 to 5) foreach ( i => println(factorial(i)) )
```

如果 fact 不是尾递归，编译器就会抛出错误。我们用这个特性在 REPL 中写出递归的 Fibonacci 函数：

```
scala> import scala.annotation.tailrec

scala> @tailrec
     | def fibonacci(i: Int): Long = {
     |   if (i <= 1) 1L
     |   else fibonacci(i - 2) + fibonacci(i - 1)
     | }

<console>:16: error: could not optimize @tailrec annotated method fibonacci:
         it contains a recursive call not in tail position
         else fibonacci(i - 2) + fibonacci(i - 1)
                               ^
```

我们有两个递归调用，然后又对调用的结果做计算，而不是只在结尾调用一次递归函数，因此这个函数不是尾递归的。

最后要说明的是，外层方法所在作用域中的一切在嵌套方法中都是可见的，包括传递给外

层方法的参数。下例 count 方法中的 n 参数就是一个例子：

```
// src/main/scala/progscala2/typelessdomore/count-to.sc

def countTo(n: Int): Unit = {
  def count(i: Int): Unit = {
    if (i <= n) { println(i); count(i + 1) }
  }
  count(1)
}
```

## 2.6 推断类型信息

静态类型语言的代码往往比较繁琐。因此我们可以考虑使用以下 Java 中的类型声明代码（Java 7 之前的版本）：

```
import java.util.HashMap;
...
HashMap<Integer, String> intToStringMap = new HashMap<Integer, String>();
```

我们不得不两次指定类型参数 <Integer, String>。Scala 使用类型注解一词表示类似 HashMap<Integer, String> 的显式类型声明。

Java 7 引入了尖括号操作符来推断表达式右边的泛型类型，降低了冗余度：

```
HashMap<Integer, String> intToStringMap = new HashMap<>();
```

我们之前已经看过一些示例说明 Scala 对类型推断确有支持。没有显式类型注解时，编译器可以根据上下文识别不少信息。利用自动推断类型信息，以上声明可以用 Scala 重写如下：

```
val intToStringMap: HashMap[Integer, String] = new HashMap
```

如果我们将 HashMap[Integer, String] 放在等号后边，代码会更简洁：

```
val intToStringMap2 = new HashMap[Integer, String]
```

一些函数式编程语言，如 Haskell，可以推断出几乎所有的类型，因为它们可以执行全局类型推断。Scala 则无法做到这一点，部分原因是 Scala 必须支持子类多态（支持继承），这使得类型推断要困难得多。

以下总结了在 Scala 中什么时候需要显式类型注解。

### 什么时候需要显式类型注解

在实际编程中，你在以下情况中必须提供显式的类型注解。

- 声明了可变的 var 变量或不可变的 val 变量，没有进行初始化。（例如，在类中的抽象声明，如 val book: String，var count: Int）。
- 所有的方法参数（如 def deposit(amount: Money) = {…}）。
- 方法的返回值类型，在以下情况中必须显式声明其类型。
  - 在方法中明显地使用了 return（即使在方法末尾也是如此）。
  - 递归方法。
  - 两个或多个方法重载（拥有相同的函数名），其中一个方法调用了另一个重载方法，调用者需要显式类型注解。
  - Scala 推断出的类型比你期望的类型更为宽泛，如 Any。

幸运的是，最后一种情况是比较少见的。

> Any 类型是 Scala 类型层次（我们会在 10.2 节介绍 Scala 的类型）中的根类型。如果一块代码意外地返回了 Any 类型的值，很有可能是因为代码比你预期的更为宽泛，于是只有 Any 类型才能覆盖所有可能的值。

我们来看几个之前没见过的例子，在这些例子中需要显式类型注释。如同以下这个示例，重载的函数中，如果其中一个调用了另一个，则需要提供显式类型注解：

```
// src/main/scala/progscala2/typelessdomore/method-overloaded-return-v1.scX
// StringUtil第一版(有一个编译错误)
// 错误:无法通过编译,右边的joiner需要一个String类型的返回值

object StringUtilV1 {
  def joiner(strings: String*): String = strings.mkString("-")

  def joiner(strings: List[String]) = joiner(strings :_*)   // 编译错误
}

println( StringUtilV1.joiner(List("Programming", "Scala")) )
```

在这个人为生造的示例中，虽然两个 joiner 方法将字符串组成的 List 连接起来，用“-”分隔，但这段代码不能通过编译。这里虽然也用了一些有用的编程惯用法，不过我们还是先解决编译错误吧。

如果你执行这段脚本，会得到下面的错误信息：

```
...
<console>:10: error: overloaded method joiner needs result type
         def joiner(strings: List[String]) = joiner(strings :_*)   // 错误
                                             ^
...
```

由于第二个 joiner 调用了第一个 joiner，第二个 joiner 就需要显式地将返回值声明为

String。应该像这样：

```
def joiner(strings: List[String]): String = joiner(strings :_*)
```

现在我们来看第二个 joiner 方法中使用的惯用技巧。Scala 支持方法拥有变量参数列表（有时也叫作“可变方法”），第一个 joiner 方法的签名为：

```
def joiner(strings: String*): String = strings.mkString("-")
```

参数列表中，String 后的 * 表示“0 个或多个 String”。方法可以拥有其他参数，但必须位于可变参数之前，且方法只能拥有一个可变参数。

然而，有时用户可能已经有了可以传递给 joiner 的字符串序列。在这种情况下，第二个 joiner 方法是一个便利的方法。它只是简单地转发调用给第一个 joiner，但它使用了特殊的语法告诉编译器将输入的 List 转为第一个 joiner 需要的可变参数列表：

```
def joiner(strings: List[String]): String = joiner(strings :_*)
```

我这么理解这段古怪的代码：strings  :_* 告诉编译器你希望列表 string 作为可变参数列表，而列表 string 的类型却不是指定的，而是根据输入推断得出的。Scala 不允许你写成 strings :String *，即使你需要为第一个 joiner 方法指定的输入类型就是 String。

事实上，并不需要这个 joiner 的便利方法，用户也可以给第一个 joiner 方法传入一个字符串列表。既然第一个 joiner 需要可变的字符串参数列表，我们可以在调用它的时候使用类似第二个 joiner 方法的语法。

对返回值类型的规定十分微妙，特别是 Scala 推断出的类型比你期望的更通用、更宽泛。将函数返回值赋给一个更具体类型的变量时，你可能会遇到这类问题。例如，你期望得到一个 String，但函数推断的返回值类型是 Any。我们再来看一个人为生造的例子，这个例子就存在以上情形所产生的 bug：

```
// src/main/scala/progscala2/typelessdomore/method-broad-inference-return.scX
// 错误:无法通过编译。方法事实上返回了List[Any],这一返回值的类型太“宽泛”了。

def makeList(strings: String*) = {
  if (strings.length == 0)
    List(0)  // #1
  else
    strings.toList
}

val list: List[String] = makeList()  // 编译错误
```

执行这段脚本会触发以下错误信息：

```
...8: error: type mismatch;
 found   : List[Any]
 required: List[String]
val list: List[String] = makeList()   // 错误
                                 ^
```

我们希望 makeList 返回一个 List[String]，但当 strings.Length 等于零时，我们返回了 List(0)，错误地“假定”这就是 Scala 中创建空列表的正确方法。实际上，我们返回的是

拥有一个元素 0 的 List[Int]。

我们应该返回 List.empty[String] 或空列表的专用产生工具 Nil。两种方法都可以得到正确的返回值推断结果。

当 if 语句返回 List[Int]，而 else 语句返回 List[String]（strings.toList 的执行结果）时，方法的返回值推断结果只能是 List[Int] 和 List[String] 的最近公共父类型，即 List[Any]。

更重要的一点是，以上代码并未产生编译错误。我们在给 list 指定了类型 List[String] 后，才在运行时发现问题。

在这个例子中，加上显式的返回类型注解 List[String] 可以在编译时就捕捉到这个错误。当然，返回类型注解也为代码的读者提供了很好的文档。

还有两个场景需要注意，省略了返回类型注解可能得到意外的推断类型。第一种，在函数中，你调用的其他函数可能会引发意外推断。这是因为该函数近期可能被修改了返回值类型。这样，你的函数在被重新编译后也会改变推断得到的返回值类型。

第二种场景常常出现在项目越来越大、不同模块构建于不同时间段的情形。如果你在集成测试中发现有 java.lang.NoSuchMethodError，或者更糟，你在生产环境实际运行时发现了这个错误，而这个被调用的方法在另一个模块中定义，这很有可能说明该函数显式地或根据推断改变了返回值类型，而调用方没有相应地进行更新，仍然在期待过时的返回类型。

开发 API 时，如果与客户端分开构建，应该显式地声明返回类型，并尽可能地返回一个你所能返回的最通用的类型。在 API 声明抽象方法（参见第 9 章）时这一点尤其重要。

最后我们用一个常见的拼写错误来结束本节。这个错误很容易犯，尤其对 Scala 新手来说。

```
scala> def double(i: Int) { 2 * i }
double: (i: Int)Unit

scala> println(double(2))
()
```

为什么第二个命令打印出了 ()，而不是 4？仔细看 scala 解释器打印的方法签名 double (Int)Unit。我们认为定义的方法名为 double，带一个 Int 参数并返回一个新的 Int 值，新的值是输入值的两倍。但它实际返回了 Uint 类型，为什么？这种意外行为产生的原因是在函数定义时漏掉了一个等号。以下才是我们想要的函数定义：

```
scala> def double(i: Int) = { 2 * i }
double: (i: Int)Int

scala> println(double(2))
4
```

注意 double 方法体之前的等号。现在，输出告诉我们，我们已经定义的 double 方法返回 Int 类型的值，第二条命令的输出也符合我们的期望。

这种现象的出现有它的原因。Scala 将方法体前的声明和等号当作函数定义，而在函数式

编程中，函数总要有返回值。另一方面，当 Scala 发现函数体之前没有等号时，就认为程序员希望该方法是一个 procedure，意味着它只返回 `Unit`。而在实践中，这很可能是因为程序员忘了写等号！

> 假如方法的返回值是推断的并且你在方法体的花括号之前没有写等号，Scala 会推断返回类型为 `Unit`，即使函数体最后一个表达式的值不是 `Unit` 类型也是如此。
>
> 这种规则太微妙，以至于很容易就会犯这种错误。由于很容易就会错误地定义一个返回 `Unit` 的方法，这种 procedure 的语法在 Scala 2.11 中已经被废除。不要用这种语法！

上文没有告诉我们 bug 修复之前输出的 `()` 是从哪来的。我们之前曾提到，`Unit` 的行为类似于其他语言的 `void`。然而，与 `void` 不同，`Unit` 这个类型拥有一个名为 `()` 的值，而这是函数式编程的一贯做法，我们将在 16.1.1 节解释它的原因。

## 2.7 保留字

表 2-1 列出了 Scala 的保留字。其中的一些我们之前已经遇到过，还有许多保留字在 Java 中也能找到，并且它们在两种语言中的含义是相同的。对于目前还没遇到的保留字，本书的后续章节会逐步涉及。

表2-1：保留字

| 保留字 | 描　述 | 参　见 |
|---|---|---|
| `abstract` | 做抽象声明 | 参见 8.1 节 |
| `case` | match 表达式中的 case 子句；定义一个 case 类 | 第 4 章 |
| `catch` | 捕捉抛出的异常 | 参见 3.9 节 |
| `class` | 声明一个类 | 参见 8.1 节 |
| `def` | 定义一个方法 | 参见 2.5 节 |
| `do` | 用于 `do...while` 循环 | 参见 3.7 节 |
| `else` | 与 `if` 配对的 `else` 语句 | 参见 3.5 节 |
| `extends` | 表示接下来的 class 或 trait 是所声明的 class 或 trait 的父类型 | 参见 8.4 节 |
| `false` | `Boolean` 的 false 值 | 参见 2.8.3 节 |
| `final` | 用于 class 或 trait，表示不能派生子类型；用于类型成员，则表示派生的 class 或 trait 不能覆写它 | 参见 11.2 节 |
| `finally` | `finally` 语句跟在相应的 `try` 语句之后，无论是否抛出异常都会执行 | 参见 3.9 节 |
| `for` | `for` 循环 | 参见 3.6 节 |
| `forSome` | 用在已存在的类型声明中，限制其能够使用的具体类型 | 参见 14.9 节 |
| `if` | `if` 语句 | 参见 3.5 节 |
| `implicit` | 使得方法或变量值可以被用于隐含转换；将方法参数标记为可选的，只要在调用该方法时，作用域内有类型匹配的候选对象，就会使用该对象作为参数 | 参见 5.3 节 |
| `import` | 将一个或多个类型抑或类型的成员导入到当前作用域 | 参见 2.12 节 |

（续）

| 保留字 | 描　　述 | 参　　见 |
|---|---|---|
| lazy | 推迟 val 变量的赋值 | 参见 3.11 节 |
| match | 用于类型匹配语句 | 第 4 章 |
| new | 创建类的一个新实例 | 参见 8.1 节 |
| null | 尚未被赋值的引用变量的值 | 参见 10.2 节 |
| object | 用于单例声明，单例是只有一个实例的类 | 参见 1.3 节 |
| override | 当原始成员未被声明为 final 时，用 override 覆写类型中的一个具体成员 | 参见 11.1 节 |
| package | 声明包的作用域 | 参见 2.11 节 |
| private | 限制某个声明的可见性 | 第 13 章 |
| protected | 限制某个声明的可见性 | 第 13 章 |
| requires | 停用，以前用于自类型 | 参见 14.6 节 |
| return | 从函数返回 | 参见 1.3 节 |
| sealed | 用于父类型，要求所有派生的子类型必须在同一个源文件中声明 | 参见 2.10 节 |
| super | 类似 this，但表示父类型 | 参见 11.3 节 |
| this | 对象指向自身的引用；辅助构造函数的方法名 | 参见 8.1 节 |
| throw | 抛出异常 | 参见 3.9 节 |
| trait | 这是一个混入模块，对类的实例添加额外的状态和行为；也可以用于声明而不实现方法，类似 Java 的 interface | 第 9 章 |
| try | 将可能抛出异常的代码块包围起来 | 参见 3.9 节 |
| true | Boolean 的 true 值 | 参见 2.8.3 节 |
| type | 声明类型 | 参见 2.13 节 |
| val | 声明一个“只读”变量 | 参见 2.2 节 |
| var | 声明一个可读可写的变量 | 参见 2.2 节 |
| while | 用于 while 循环 | 参见 3.7 节 |
| with | 表示所声明的类或实例化的对象包括后面的 trait | 第 9 章 |
| yield | 在 for 循环中返回元素，这些元素会构成一个序列 | 参见 3.6.4 节 |
| _ | 占位符，使用在 import、函数字面量中 | 很多章节均涉及 |
| : | 分隔标识符和类型注解 | 参见 1.3 节 |
| = | 赋值 | 参见 1.3 节 |
| => | 在函数字面量中分隔参数列表与函数体 | 参见 6.2.1 节 |
| <- | 在 for 循环中的生成表达式 | 参见 3.6 节 |
| <: | 在参数化类型和抽象类型声明中，用于限制允许的类型 | 参见 14.2 节 |
| <% | 在参数化类型和抽象类型的 view bound 声明中 | 参见 14.4 节 |
| >: | 在参数化类型和抽象类型声明中，用于限制允许的类型 | 参见 14.2 节 |
| # | 在类型注入中使用 | 参见 15.3 节 |
| @ | 注解 | 参见 23.2 节 |
| ⇒ | (Unicode \u21D2)，与 => 相同 | 参见 6.2.1 节 |
| → | (Unicode \u2192)，与 -> 相同 | 参见 5.3 节 |
| ← | (Unicode \u2190)，与 <- 相同 | 参见 3.6 节 |

注意，表中没有列出 `break` 和 `continue`。这两个流程控制的关键字在 Scala 中不存在。Scala 鼓励使用函数式编程的惯用法来实现相同的 `break`、`continue` 功能。函数式编程通常会更加简洁，不容易出现 bug。

一些 Java 中的方法名在 Scala 中是保留字。如 `java.util.Scanner.match`。为了避免编译错误，引用该方法名时，在名字两边加上反引号，如 ``java.util.Scanner.`match` ``。

# 2.8 字面量

我们在前面已经遇到过一些字面量，如 `val book = "Programming Scala"`。在这行代码中，我们用一个 `String` 字面量初始化了一个 `val` 变量 `book`。还有 `(s: String) => s.toUpperCase`，这也是一个函数字面量的例子。我们现在来讨论 Scala 支持的所有字面量。

## 2.8.1 整数字面量

整数字面量可以以十进制、十六进制或八进制的形式出现。表 2-2 给出了整数字面量的详细信息。

**表2-2：整数字面量**

| 类　型 | 格　式 | 例　子 |
|---|---|---|
| 十进制 | 0 或一个非零值，后面跟上 0 个或多个数字（0-9） | 0,1,321 |
| 十六进制 | 0x 后面跟上一个或多个十六进制数字（0-9，A-F，a-f） | 0xFF,0x1a3b |
| 八进制 | 0 后面跟上一个多个八进制数字（0-7）[a] | 013,077 |

a 八进制数字字面量在 Scala 2.10 中已经废弃。

在数字字面量之前加上 – 符号，可以表示负数。

对于 `Long` 类型的字面量，除非你将该字面量赋值给一个 `Long` 类型的变量，否则需要在数字字面量后面加上 `L` 或 `l`。如若不加，字面量的类型将默认推断为 `Int`。整数字面量的有效值范围是根据被赋值的变量类型决定的。表 2-3 列出了各类型的上下限（包含上下限本身）。

**表2-3：整数字面量的取值范围（包含边界）**

| 目标类型 | 下限（包含） | 上限（包含） |
|---|---|---|
| `Long` | $-2^{63}$ | $2^{63}-1$ |
| `Int` | $-2^{31}$ | $2^{31}-1$ |
| `Short` | $-2^{15}$ | $2^{15}-1$ |
| `Char` | 0 | $2^{16}-1$ |
| `Byte` | $-2^{7}$ | $2^{7}-1$ |

如果整数字面量的值超出了以上表格中所示的范围，将会引发一个编译错误。如下面这个例子：

```
scala> val i = 1234567890123
<console>:1: error: integer number too large
       val i = 12345678901234567890
```

```
                ^

scala> val i = 1234567890123L
i: Long = 1234567890123

scala> val b: Byte = 128
<console>:19: error: type mismatch;
 found   : Int(128)
 required: Byte
       val b: Byte = 128
                     ^

scala> val b: Byte = 127
b: Byte = 127
```

### 2.8.2 浮点数字面量

浮点数字面量的形式为：首先是一个可省略的负号，然后是 0 个或多个数字后面跟上一个点号（.），后面再跟上一个或多个数字。对于 Float 类型的字面量，还要在后面加上 F 或 f，否则会被认为是 Double 类型。你也可以在后面加上 D 或 d，明确表示字面量为 Double 字面量。

浮点数字面量可以用指数表示。指数部分的形式为 E 或 e 后面跟上可选的 + 或 -，然后是一个或多个数字。

以下是浮点数字面量的一些例子。例子中除非被赋值的变量是 Float 类型，或者在字面量后面加了 F 或 f，否则字面量都被推断为 Double 类型：

```
.14
3.14
3.14f
3.14F
3.14d
3.14D
3e5
3E5
3.14e+5
3.14e-5
3.14e-5
3.14e-5f
3.14e-5F
3.14e-5d
3.14e-5D
```

Float 遵循 IEEE 754 32 位单精度浮点数规范。

Double 遵循 IEEE 754 64 位双精度浮点数规范。

在 Scala 2.10 之前，小数点后没有数字是允许的，如 3. 和 3.e5。但是由于这种语法会导致歧义：小数点可能被解释为方法名前的句号，因此 1.toString 应该如何解析？是 Int 类型的 1，还是 Double 类型的 1.0？所以，小数点后不带数字的浮点数字面量在 Scala 2.10 中就被废弃，而在 Scala 2.11 中则不被允许。

## 2.8.3　布尔型字面量

布尔型字面量可以为 true 或 false。被这两个字面量赋值的变量，其类型将被推断为 Boolean：

```
scala> val b1 = true
b1: Boolean = true

scala> val b2 = false
b2: Boolean = false
```

## 2.8.4　字符字面量

字符字面量要么是单引号内的一个可打印的 Unicode 字符，要么是一个转义序列。值在 0~255 的 Unicode 字符也可以用八进制数字的转义形式表示，即一个反斜杠（\）后面跟上最多 3 个八进制数字字符。如果反斜杠后面不是有效的转义序列，会引发编译时的错误。

以下是字符字面量的例子：

```
'A'
'\u0041'  // Unicode中的'A'
'\n'
'\012'    // 八进制的'\n'
'\t'
```

表 2-4 给出了有效的转义序列。

表2-4：字符转义序列

| 转义序列 | 含　　义 |
|---|---|
| \b | 退格（BS） |
| \t | 水平制表符（HT） |
| \n | 换行（LF） |
| \f | 表格换行（FF） |
| \r | 回车（CR） |
| \" | 双引号（"） |
| \’ | 单引号（’） |
| \\ | 反斜杠（\） |

注意，不可打印的 Unicode 字符，如 \u0009（水平制表符）是不允许的。应该使用等价的转义形式 \t。回顾一下表 2-1 中提到的 3 个 Unicode 字符，可以有效地替换相应的 ASCII 序列：⇒ 替换 =>，→替换 ->，←替换 <-。

## 2.8.5　字符串字面量

字符串字面量是被双引号或者三重双引号包围的字符串序列，如 """…"""。

对于双引号包围的字符串字面量，允许出现的字符与字符字面量相同。但是，如果双引号

字符出现在字符串中，则必须使用反斜杠 \ 进行转义。以下是字符串字面量的例子：

```
"Programming\nScala"
"He exclaimed, \"Scala is great!\""
"First\tSecond"
```

用三重双引号包含的字符串字面量也被称为多行字符串字面量。这些字符串可以跨越多行，换行符是字符串的一部分。可以包含任意字符，可以是一个双引号也可以是两个连续的双引号，但不允许出现三个连续的双引号。对于包含有反斜杠 \，但反斜杠不用于构成 Unicode 字符，也不用于构成有效转义序列（如表 2-4 中列出的序列）的字符串很适合采用这种字符串字面量。正则表达式就是一个很好的例子，在正则表达式中经常用转义的字符表示特殊含义。即使转义序列出现，三重双引号包含的字符串也不对其进行转义。

以下给出了 3 个示例字符串：

```
"""Programming\nScala"""
"""He exclaimed, "Scala is great!" """
"""First line\n
Second line\t

Fourth line"""
```

注意，在第二个例子中，必须在结尾的 `"""` 前加一个空格，以防止出现解析错误。试图将 `"Scala is great!"` 的第二个双引号进行转义（`"Scala is great!\"`）的行为是无效的。

使用多行字符串时，你可能希望各行子串有良好的缩进以使代码美观，但却不希望多余的空格出现在字符串的输出中。`String.stripMargin` 可以解决这个问题。它会移除每行字符串开头的空格和第一个遇到的垂直分隔符 `|`。如果你需要自行添加空格制造缩进，你应该在 `|` 后添加你要的空格。参照以下示例：

```
// src/main/scala/progscala2/typelessdomore/multiline-strings.sc

def hello(name: String) = s"""Welcome!
  Hello, $name!
  * (Gratuitous Star!!)
  |We're glad you're here.
  |  Have some extra whitespace.""".stripMargin

hello("Programming Scala")
```

以上代码输出如下：

```
Welcome!
  Hello, Programming Scala!
  * (Gratuitous Star!!)
We're glad you're here.
  Have some extra whitespace.
```

注意哪些行开头的空格被移除，而哪些行开头的空格未被移除。

如果你希望用别的字符代替 `|`，可以用 `stripMargin` 的重载版本，该函数可以指定一个 `Char` 参数代替 `|`。如果你想要移除整个字符串（而不是字符串的各个行）的前缀和后缀，有相应的 `stripPrefix` 和 `stripSuffix` 方法可以完成：

```
// src/main/scala/progscala2/typelessdomore/multiline-strings2.sc

def goodbye(name: String) =
  s"""xxxGoodbye, ${name}yyy
  xxxCome again!yyy""".stripPrefix("xxx").stripSuffix("yyy")

goodbye("Programming Scala")
```

上述例子的输出为：

```
Goodbye, Programming Scalayyy
  xxxCome again!
```

### 2.8.6 符号字面量

Scala 支持符号，符号是一些规定的字符串。两个同名的符号会指向内存中的同一对象。相比其他编程语言如 Ruby 和 Lisp，符号在 Scala 中用得比较少。

符号字面量是单引号（'）后跟上一个或多个数字、字母或下划线（“_”），但第一个字符不能为数字。所以 '1symbol 是无效的符号。

符号字面量 *'id* 是表达式 scala.Symbol("id") 的简写形式，如果你需要创建一个包含空格的符号，可以使用 Symbol.apply，如 Symbol(" Programming Scala ") 中的空格均为符号的一部分。

### 2.8.7 函数字面量

我们之前已经接触过函数字面量，但这里重述一下：(i: Int, s: String) => s+i 是一个类型为 Function2[Int,String,String]（返回值类型为 String）的函数字面量。

甚至可以用函数字面量来声明变量，以下两种声明是等价的：

```
val f1: (Int,String) => String       = (i, s) => s+i
val f2: Function2[Int,String,String] = (i, s) => s+i
```

### 2.8.8 元组字面量

你希望从方法中返回多少次值（两个或多个值）？在很多语言中，如 Java，你只有少数几种解决方案，且每一种都不是上选。比如，你可以传入参数，然后在方法中修改该参数，相当于“返回值”，但这样的代码很丑陋，不美观。（有的语言甚至用关键字来表示哪些参数是输入参数，哪些参数是输出参数。）你可以声明一个像“结构体”一样的类，类中有两个或多个值，然后返回该类的实例。

Scala 库中包含 TupleN 类（如 Tuple2），用于组建 *N* 元素组，它以小括号加上逗号分隔的元素序列的形式来创建元素组。TupleN 表示的多个类各自独立，*N* 的取值从 1 到 22，包括 22（在 Scala 的未来版本中这个上限可能最终取消）。

例如，val tup = ("Programming Scala", 2014) 定义了一个 Tuple2 的实例，实例中的第一个成员类型为 String，从 "Programming Scala" 中推断得到，第二个成员类型为 Int，从

2014 中推断得到。元组的实例是不可变的、first-class 的值（因为它们是对象，与你定义的其他类的实例没有区别），所以你可以将它们赋值给变量，将它们作为输入参数，或从方法中将其返回。

你也可以用字面量语法来声明 Tuple 类型的变量：

```
val t1: (Int,String)       = (1, "two")
val t2: Tuple2[Int,String] = (1, "two")
```

以下例子演示了元组的用法：

```
// src/main/scala/progscala2/typelessdomore/tuple-example.sc

val t = ("Hello", 1, 2.3)                                         // ❶
println( "Print the whole tuple: " + t )
println( "Print the first item:  " + t._1 )                       // ❷
println( "Print the second item: " + t._2 )
println( "Print the third item:  " + t._3 )

val (t1, t2, t3) = ("World", '!', 0x22)                           // ❸
println( t1 + ", " + t2 + ", " + t3 )

val (t4, t5, t6) = Tuple3("World", '!', 0x22)                     // ❹
println( t4 + ", " + t5 + ", " + t6 )
```

❶ 用字面量语法构造一个三个参数的元组 Tuple3。

❷ 从元组中提取第一个元素（计数从 1 开始，不从 0 开始），接下来的 2 行代码分别提取第二个和第三个元素。

❸ 声明了三个变量，t1、t2、t3，用元组中三个对应的元素对其赋值。

❹ 用 Tuple3 的“工厂”方法构造一个元组。

运行该脚本，得到以下输出：

```
Print the whole tuple: (Hello,1,2.3)
Print the first item:  Hello
Print the second item: 1
Print the third item:  2.3
World, !, 34
World, !, 34
```

表达式 t._n 提取元组 t 中的第 *n* 个元素。为遵循历史惯例，这里从 1 开始计数，而不是从 0 开始。

一个两元素的元组，有时也被简称为 pair。有很多定义 pair 的方法，除了在圆括号中列出元素值以外，还可以把“箭头操作符”放在两个值之间，也可以用相应类的工厂方法：

```
(1, "one")
1 -> "one"
1 → "one"          // 用 → 代替 ->
Tuple2(1, "one")
```

箭头操作符只适用于两元素的元组。

## 2.9 Option、Some和None：避免使用null

我们来讨论 3 种类型，即 Option、Some 和 None，它们可以表示“有值”或者“没有值”。

大部分语言都有一个特殊的关键字或类的特殊实例，用于在引用变量没有指向任何对象时，表示“无”。在 Java 中，是 null 关键字，但 Java 中没有某个实例或类型。因此，对它调用任何方法都是非法的。但是语言设计者对此感到非常迷惑：为什么要在程序员期望返回对象时返回一个关键字呢？

当然，真正的问题在于，null 是很多 bug 的来源。null 表示的真正含义是在给定的情形下没有任何值。如何变量不等于 null，它是有值的。为什么不在类型系统中显式地将这种情况表达出来，并通过类型检查来避免空指针异常呢？

Option（http://www.scala-lang.org/api/current/#scala.Option）允许我们显式表示这种情况，而不需要用 null 这种“骇客”技巧。作为一个抽象类，Option 却有两个具体的子类 Some（http://www.scala-lang.org/api/current/#scala.Some）和 None（http://www.scala-lang.org/api/current/#scala.None$）。Some 用于表示有值，None 用于表示没有值。

在以下实例中你会看到 Option、Some 和 None 的运用。我们创建了一个美国州与州首府的映射表：

```
// src/main/scala/progscala2/typelessdomore/state-capitals-subset.sc

val stateCapitals = Map(
  "Alabama" -> "Montgomery",
  "Alaska"  -> "Juneau",
  // ...
  "Wyoming" -> "Cheyenne")

println( "Get the capitals wrapped in Options:" )
println( "Alabama: " + stateCapitals.get("Alabama") )
println( "Wyoming: " + stateCapitals.get("Wyoming") )
println( "Unknown: " + stateCapitals.get("Unknown") )

println( "Get the capitals themselves out of the Options:" )
println( "Alabama: " + stateCapitals.get("Alabama").get )
println( "Wyoming: " + stateCapitals.get("Wyoming").getOrElse("Oops!") )
println( "Unknown: " + stateCapitals.get("Unknown").getOrElse("Oops2!") )
```

注意看执行脚本时发生了什么：

```
Get the capitals wrapped in Options:
Alabama: Some(Montgomery)
Wyoming: Some(Cheyenne)
Unknown: None
Get the capitals themselves out of the Options:
Alabama: Montgomery
Wyoming: Cheyenne
Unknown: Oops2!
```

Map.get 方法返回了 Option[T]，这里类型 T 为 String。与此不同，在 Java 中，Map.get 返回 T，T 可能是 null 或实际的值。通过返回 Option，我们就不会“忘记”去检查是否有实

际值返回。换言之，对于给定的 key，“对应的值可能并不存在”这一事实已经包含在方法返回的类型中了。

第一组 println 语句非直接地对 get 返回的实例执行了 toString 方法。事实上，我们是在对 Some 或 None 执行 toString 方法，因为当映射表中存在 key 对应的值时，Map.get 的返回值被自动包装在 Some 对象中。相反，当我们请求了一个映射表中不存在的数据时，Map.get 就返回 None，而不是 null，最后一个 println 就是这种情况。

第二组 println 更进一步，调用 Map.get 后，又对 Option 实例调用了 get 或 getOrElse，以取出其中包含的值。

Option.get 方法有些危险，如果 Option 是一个 Some，Some.get 则会返回其中的值。然而，如果 Option 事实上是一个 None，None.get 就会抛出一个 NoSuchElementException（http://docs.oracle.com/javase/8/docs/api/java/util/NoSuchElementException.html）异常。

在后面两个 println 语句中，我们可以看到 get 的替代选项——一个更安全的方法 getOrElse。getOrElse 方法会在 Option 为 Some 时返回其中的值，而在 Option 为 None 时返回传递给它的参数中的值。换言之，getOrElse 的参数起到了默认值的作用。

所以，getOrElse 是两个方法中更具防御性的，它避免了潜在的异常。

再次申明，由于 Map.get 返回了 Option，这明显告诉读者映射表中有可能找不到指定的 key。映射表在这种情况下会返回 None，而大部分编程语言会返回 null（或其他等价的值）。的确，从经验来看你推断这种情况下这些语言中可能出现 null，但 Option 将这种情况显式地体现在函数签名中，使其更具自解释性。

另外，多亏 Scala 的静态类型性质，你可以避免“忘记”返回的是 Option，从而调用 Option 里的值（如果有值的话）来启动方法。在 Java 中，当方法返回一个值时，在调用该值的方法前很容易忘记检查它是否为 null。但是，Scala 的方法返回 Option，编译器的类型检查便强制要求你先从 Option 中提取值，再对它调用方法。这一机制“提醒”你去检查 Option 是否等于 None。所以，Option 的使用强烈鼓励更具弹性的编程习惯。

Scala 运行于 JVM 的环境，且必须与其他库互操作，因此 Scala 必须支持 null。另外，一些固执的人也可能会给你返回一个 Some(null)。尽管如此，你现在有了比 null 更好的选择，就应该在自己的代码中避免使用 null（除非必须与支持 null 的 Java 库进行交互）。Tony Hoare（http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare），在 1965 年开发一门名为 ALGOL W 的语言时发明了 null，他曾表示 null 的发明是相当于损失“数十亿美元”的错误。你看，还是用 Option 吧。

## 2.10 封闭类的继承

现在我们来探讨 Option 的一个很有用的特性。Option 的一个关键点在于它只有两个有效的子类。如果我们有值，则对应 Some 子类；如果没有值，则对应 None 子类。没有其他有效的 Option 子类型。所以，我们可以防止用户创建一个他们自己的子类。

为了达到这个目的，Scala 设计了关键字 sealed。Option 的声明类似这样（省略部分细节）：

```
sealed abstract class Option[+A] ... { ... }
```

关键字 sealed 告诉编译器，所有的子类必须在同一个源文件中声明。而在 Scala 库中，Some 与 None 就是与 Option 声明在同一源文件中的。这一技术有效防止了 Option 派生其他子类型。

顺便提一下，如果需要防止用户派生任何子类，也可以用 final 关键字进行声明。

## 2.11 用文件和名空间组织代码

Scala 沿用 Java 用包来表示命名空间的这一做法，但它却更具灵活性。文件名不必与类名一致，包结构不一定要与目录结构一致。所以，你可以定义与文件的“物理”位置独立的包结构。

以下示例用 Java 语法在包 com.example.mypkg 中定义了名为 MyClass 的类，这是 Java 的常规语法：

```
// src/main/scala/progscala2/typelessdomore/package-example1.scala
package com.example.mypkg

class MyClass {
  // ...
}
```

Scala 也支持使用嵌套块结构语法来定义包的作用域，与 C# 的命名空间语法和 Ruby 表示命名空间的 modules 用法类似：

```
// src/main/scala/progscala2/typelessdomore/package-example2.scala
package com {
  package example {
    package pkg1 {
      class Class11 {
        def m = "m11"
      }
      class Class12 {
        def m = "m12"
      }
    }

    package pkg2 {
      class Class21 {
        def m = "m21"
        def makeClass11 = {
          new pkg1.Class11
        }
        def makeClass12 = {
          new pkg1.Class12
        }
      }
    }

    package pkg3.pkg31.pkg311 {
```

```
        class Class311 {
          def m = "m21"
        }
      }
    }
  }
```

pkg1 和 pkg2 这两个包定义在包 com.example 中。在这两个包中，共有 3 个类。在包 pkg2 的类 Class21 中，makeClass11 和 makeClass12 方法展示了如何引用“兄弟”包 pkg1 中的类。你也可以分别用这些类的全路径 com.example.pkg1.Class11 和 com.example.pkg1.Class12 来引用它们。

包 pkg3.pkg31.pkg311 表明你可以在一条语句中将多个包“链接”在一起。不必为每一层包单独使用一条 package 语句。

然而，有一种情况你必须使用单独的 package 语句。我们称之为连续包声明：

```
// src/main/scala/progscala2/typelessdomore/package-example3.scala
// 导入example中所有包级别的声明
package com.example
// 导入mypkg中所有包级别的声明
package mypkg

class MyPkgClass {
  // ...
}
```



如果你想导入的包都有包级别的声明，比如类的声明，那么应该为包层次中的每个包使用单独的 package 语句。每个后续的包语句都被解释为上一个包的子包，就像我们使用上文展示的嵌套块结构语法那样。第一个 package 语句的路径为绝对路径。

我们遵循 Java 的惯例，将 Scala 库的 root 包命名为 scala。

尽管声明包的语法很灵活，但有一个限制，就是包不能在类或对象中定义，那样做是没有意义的。

Scala 不允许在脚本中定义包，脚本被隐含包装在一个对象中。在对象中声明包是不允许的。

## 2.12 导入类型及其成员

就像在 Java 中一样，要使用包中的声明，必须先导入它们。但 Scala 还提供了其他选择，以下例子展示了 Scala 如何导入 Java 类型：

```
import java.awt._
import java.io.File
import java.io.File._
import java.util.{Map, HashMap}
```

你可以像第一行那样，用下划线（_）当通配符，导入包中的所有类型。你也可以像第二行那样导入包中单独的 Scala 类型或 Java 类型。

> Java 用星号（*）作为 import 的通配符。在 Scala 中，星号被允许用作函数名，因此 _ 被用作通配符，以避免歧义。例如，如果对象 `Foo` 定义了其他方法，同时它还定义了 * 方法，`import Foo.*` 表示什么呢？

第三行导入了 `java.io.File` 中所有的静态方法和属性。与之等价的 Java import 语句为 `import static java.io.File.*`。Scala 没有 `import static` 这样的写法，因为 Scala 将 `object` 类型与其他类型一视同仁。

如第四行所示，选择性导入的语法非常好用，在第四行中我们导入了 `java.util.Map` 和 `java.util.HashMap`。

import 语句几乎可以放在任何位置上，所以你可以将其可见性限制在需要的作用域中，可以在导入时对类型做重命名，也可以限制不想要的类型的可见性：

```
def stuffWithBigInteger() = {

  import java.math.BigInteger.{
    ONE => _,
    TEN,
    ZERO => JAVAZERO }

  // println( "ONE: "+ONE )     // ONE未定义
  println( "TEN: "+TEN )
  println( "ZERO: "+JAVAZERO )
}
```

由于这一 import 语句位于 `stuffWithBigInteger` 函数中，导入的类型和值在函数外是不可见的。

将 `java.math.BigInteger.ONE` 常量重命名为下划线，使得该常量不可见。当你需要导入除少部分以外的所有声明时，可以采用这一技术。

接着，`java.math.BigInteger.TEN` 导入时未经重命名，所以可以用 `TEN` 引用它。

最后，`java.math.BigInteger.ZERO` 常量被赋予了 `JAVAZERO` 的“别名”。

当你想取一个便利的名字或避免与当前作用域中其他同名声明冲突时，别名非常有用。导入 Java 类型时经常使用别名，以避免其余 Scala 中同名类型的冲突，如 `java.util.List`（http://docs.oracle.com/javase/8/docs/api/java/util/List.html）与 `java.util.Map`（http://docs.oracle.com/javase/8/docs/api/java/util/Map.html），在 Scala 库中有相同名称的类。

## 2.12.1 导入是相对的

Scala 与 Java 导入机制的另一重要区别是：Scala 的导入是相对的。注意下例中的注释：

```
// src/main/scala/progscala2/typelessdomore/relative-imports.scala
import scala.collection.mutable._
```

```
import collection.immutable._            // 由于scala已经导入,不需要给出全路径
import _root_.scala.collection.parallel._  // 从"根"开始的全路径
```

在相对导入上很少会碰见麻烦，但意外有时也会发生。如果你遇到一个让你迷惑不解的编译错误指出某个包无法找到时，需要检查一下导入语句中的相对路径和绝对路径是否正确。在少数情况下，你还需要添加 _root_ 前缀。通常，使用顶层的包，如 com、org 或 scala 就足够了。但必须保证问题库的所在路径被包含在 CLASSPATH 中。

## 2.12.2 包对象

对于库的作者，设计上要选择何处作为 API 暴露公共接入点，因为这些 API 是客户端将要导入并使用的。Java 库经常导入包中定义的部分或所有类型。例如 Java 语句 import java.io.* 导入了 io 包中的所有类型。Java 5 增加了"静态导入"，支持单独导入包中的静态成员。尽管相对便利了一些，但它所采用的语法还是不太便利。你可以考虑使用一个虚构的 JSON 解析库，同时会用到顶层的库 json 以及类 JSON 中的一个静态 API 方法：

```
static import com.example.json.JSON.*;
```

在 Scala 中，你至少可以省略 static 关键字。但写为如下形式更为简洁，可以用一条 import 语句导入所有 API 使用者需要的类型、方法和值：

```
import com.example.json._
```

Scala 支持包对象这一特殊类型的、作用域为包层次的对象。这里的 json 就是包对象。它像普通的对象一样声明，但与普通对象有着如下示例所展示的不同点：

```
// src/com/example/json/package.scala                    ❶

package com.example                                     // ❷

package object json {                                   // ❸
  class JSONObject {...}                                // ❹
  def fromString(string: String): JSONObject = {...}
  ...
}
```

❶ 文件名必须为 package.scala。根据惯例，文件位于与定义的包对象相同的包目录中，在这里为 com/example/json。

❷ 上层包的作用域。

❸ 使用 package 关键字给包名之后的对象命名，在这里对象名为 json。

❹ 适合暴露给客户端的成员。

这样，客户端可以用 import com.example.json._ 导入所有的定义，或用通常的方法单独导入元素。

# 2.13 抽象类型与参数化类型

我们在 1.3 节中提到 Scala 支持参数化类型，与 Java 中的泛型十分类似。（我们也可以交换

这两个术语，但 Scala 社区中多使用“参数化类型”，Java 社区中常用泛型一词。）在语法上，Java 使用尖括号（<…>），而 Scala 使用方括号（[…]），因为在 Scala 中 < 和 > 常用作方法名。

例如，字符串列表可以声明如下：

```
val strings: List[String] = List("one", "two", "three")
```

由于我们可以在集合 List[A] 中使用任何类型作为类型 A，这种特性被称为参数多态。在方法 List 的通用实现中，允许使用任何类型的实例作为 List 的元素。

我们来讨论学习参数化类型最为重要的细节，尤其当你在试图理解 Scaladoc 中的类型签名时，这些细节尤为重要。如在 Scaladoc 中的 List 条目（http://www.scala-lang.org/api/current/#scala.collection.immutable.List）中，你会发现其声明被写为 sealed abstract class List[+A]。

A 之前的 + 表示：如果 B 是 A 的子类，则 List[B] 也是 List[A] 的子类型，这被称为协类型。协类型很符合直觉，如果我们有一个函数 f(list: List[Any])，那么传递 List[String] 给这个函数，也应该能正常工作。

如果类型参数前有 -，则表示另一种关系：如果 B 是 A 的子类型，且 Foo[A] 被声明为 Foo[-A]，则 Foo[B] 是 Foo[A] 的父类型（称为逆类型）。这一机制没那么符合直觉，我们将在参数化类型中与参数化类型的其他细节一起解释这一点。

Scala 还支持另一种被称为“抽象类型”的抽象机制，它可以运用在许多参数化类型中，也能够解决设计上的问题。然而，尽管两种机制有所重合，但并不冗余，两种机制对不同的设计问题各有优势与不足。

参数化类型和抽象类型都被声明为其他类型的成员，就像是该类型的方法与属性一样。以下示例在父类中使用抽象类型，而在子类中将该类型具体化：

```
// src/main/scala/progscala2/typelessdomore/abstract-types.sc
import java.io._

abstract class BulkReader {
  type In
  val source: In
  def read: String  // 读进source,然后返回一个String
}

class StringBulkReader(val source: String) extends BulkReader {
  type In = String
  def read: String = source
}

class FileBulkReader(val source: File) extends BulkReader {
  type In = File
  def read: String = {
    val in = new BufferedInputStream(new FileInputStream(source))
    val numBytes = in.available()
    val bytes = new Array[Byte](numBytes)
    in.read(bytes, 0, numBytes)
```

```
      new String(bytes)
    }
  }

  println(new StringBulkReader("Hello Scala!").read)
  // 假定当前目录为src/main/scala:
  println(new FileBulkReader(
    new File("TypeLessDoMore/abstract-types.sc")).read)
```

产生的输出如下：

```
Hello Scala!
// src/main/scala/progscala2/typelessdomore/abstract-types.sc

import java.io._

abstract class BulkReader {
...
```

抽象类 BulkReader 声明了 3 个虚拟成员：一个名为 In，是类型成员；第二个类型为 In，是 val 变量，名为 source；第三个是一个 read 方法。

派生类 StringBulkReader 与 FileBulkReader 为上述抽象成员提供具体化的定义。

注意 type 成员的工作机制与参数化类型中的类型参数非常类似。事实上，我们可以将该示例重写如下，在这里我们只显示改动的部分：

```
abstract class BulkReader[In] {
  val source: In
  ...
}

class StringBulkReader(val source: String) extends BulkReader[String] {...}

class FileBulkReader(val source: File) extends BulkReader[File] {...}
```

就像参数化类型，如果我们定义 In 类型为 String，则 source 属性也必须被定义为 String。注意 StringBulkReader 的 read 方法只是将 source 属性返回，而 FileBulkReader 的 read 方法则需要读取文件的内容。

那么，类型成员与参数化类型相比有什么优势呢？当类型参数与参数化的类型无关时，参数化类型更适用。例如 List[A]，A 可能是 Int、String 或 Person 等。而当类型成员与所封装的类型同步变化时，类型成员最适用。正如 BulkReader 这个例子，类型成员需要与封装的类型行为一致。有时这种特点被称为家族多态，或者协特化。

## 2.14 本章回顾与下一章提要

本章介绍了 Scala 编程的实践基础，如字面量、关键字、文件的组织以及导入。我们学习了如何声明变量、方法和类，也了解了 Option 是比 null 更好用的工具，以及其他一些有用的技术。在下一章中，我们将结束对 Scala 基础的概览，深入解释 Scala 特性的细节。

# 第 3 章

# 要点详解

本章将讲解必要的 Scala 基础知识。

## 3.1 操作符重载?

在 Scala 中，几乎所有的“操作符”其实都是方法。我们一起看看最基础的一个例子：

```
1 + 2
```

数字之间的加号是什么呢？这个操作符是一个方法。

首先，请注意在 Scala 的世界里，Java 中特殊的“基本类型”都变成了正规的对象，这些对象类型为：`Float`、`Double`、`Int`、`Long`、`Short`、`Byte` 和 `Boolean` 类型。这也意味着它们可以拥有成员方法。

正如你所见，除了某些特殊情况之外，Scala 标示符中允许出现字母和数字之外的字符。我们稍后将会讲到这些特殊情况。

由于使用中缀表示法表示单参数方法时，其中的点号和括号可以省略，因此 `1 + 2` 等价于 `1.+(2)`。[1]

与之相似，调用无参方法时也可以省略点号。这种写法也被称为后缀表示法。不过有时候使用后缀表示方式时会产生歧义，因此 Scala 2.10 将这种表示法修改为可选特性。我们将搭建 SBT 的实验环境，假如你在没有告诉编译器的情况下使用了该特性，那么将触发一条

注 1：实际上根据操作符优先级规则，包含点号和省去点号的表达式并不总是完全一致的。`1 + 2 * 3 = 7`，而 `1.+(2)*3 = 9`。如果表达式中包含点号，那么优先执行点号对应的操作，再执行乘法运算。另外如果你使用了 2.11 版本之前的 Scala，请在数字 `1` 后添加一个空格，否则 `1` 将会被解析为 `Double` 类型！

警告信息。可以通过一条 import 语句开启此特性。请查看下面的示例了解通过 `scala` 命令开启 REPL 会话（也可以通过 SBT `console` 命令开启）。

```
$ scala
...
scala> 1 toString
Warning: there were 1 feature warning(s); re-run with -feature for details
res0: String = 1
```

直接运行 `scala` 命令看上去并不能启动此特性，我们加上 `-feature` 标志重新启动 REPL，以获取更多有意义的警告信息。

```
$ scala -feature
...
scala> 1.toString  // normal invocation
res0: String = 1

scala> 1 toString  // postfix inovocation
<console>:8: warning: postfix operator toString should be enabled
by making the implicit value scala.language.postfixOps visible.
This can ... adding the import clause 'import scala.language.postfixOps'
or by setting the compiler option -language:postfixOps.
See the Scala docs for value scala.language.postfixOps for a discussion
why the feature should be explicitly enabled.
              1 toString
                ^
res1: String = 1

scala> import scala.language.postfixOps
import scala.language.postfixOps

scala> 1 toString
res2: String = 1
```

我希望任何时候出现了这样的问题时，都能看到这个详细的错误信息，所以我进行了 SBT 项目的配置，启用了 `-feature` 标志。这样一来，在 SBT 中使用 `console` 任务运行 REPL 时，这个标志便已开启了。

可以通过两种方式消除这个警告。我们已经演示了第一种方法：执行 `import scala.language.postfixOps` 命令。我们也可以向编译器传递另一个标志 `-language:postfixOps`，通过这一方式在全局范围内开启该特性。我个人青睐于每次都调用 import 语句，因为这种方式能提醒读者了解到我们现在启用了哪个可选特性。（我们会在 21.1.1 节列出所有可选特性。）

在不造成编译器歧义的前提下，省略点号能够使代码变得更加整洁，也有助于创建更优雅、更自然易懂的程序。

那么，哪些字符允许出现在标示符中呢？下面总结了方法名、类型名、变量名等各种标示符需要遵循的规则。

- 可用的字符

  除了括号类字符、分隔符之外，其他所有的可打印的 ASCII 字符如字母、数字、下划

线（_）和美元符号（$）均可出现在 Scala 标示符中，插入字符包括了(,)、[,]、{, and}；而分隔符则包括了' 、’、'、"、.、; 以及 ,。Scala 还允许在标示符中使用编码在 \u0020 到 \u007F 之间的字符，如数学符号、像 / 和 < 这样的操作符字符以及其他的一些符号。

- 不能使用保留字

  和绝大多数语言一样，Scala 中也不允许使用保留字作为标示符。我们将在 2.7 节列举所有的保留字。我们回忆一下，某些操作符和标点符号也属于保留字，例如，下划线（_）便是一个保留字。

- 普通标示符——字母、数字、$、_ 和操作符的组合

  常见的标示符往往由字母或下划线开头，后面跟着一些字母、数字、下划线或美元符。Scala 允许使用 Unicode 格式的字符。由于美元符在 Scala 内部会作为特定作用，因此尽管编译器不会阻止你，你仍不应将美元符当作标示符使用。在下划线后可以输入字母、数字，也可以输入一些操作符。下划线就一个重要的字符，编译器会将下划线之后空格之前的所有字符视为标示符的一部分。举例而言，`val xyz_++= = 1` 会将变量 `xyz_++` 赋值为 `1`，而表达式 `val xyz++= =1` 则无法通过编译。这是因为标示符 `xyz++=` 也可以被解释为 `xyz ++=`，这看上去像是要为 `xyz` 赋值。出于同样的原因，假如在下划线后输入了操作符，那么不允许在操作符后输入字母或数字。这一限制确保了不会产生具有歧义的表达式，例如：`abc_-123`。该语句到底是表示标示符 `abc_-123`，还是变量 `abc_` 减去 `123` 呢？

- 普通标示符——操作符

  假如某一标示符以操作符开头，那么后面的字符也必须是操作符字符。

- 使用反引号定义标示符

  我们可以通过反引号定义标示符，两个反引号内的任意长度的字符串便是定义的标示符，``def`test that addition works`= assert(1 + 1 == 2)`` 便是其中一例（这句代码应用反引号定义测试名称的方法，这种方式使用了一种“如果不满足某些条件，便存在问题”的命名技巧。而你将来也许能在产品代码中看到这类命名方式）。我们之前也曾看到过反引号命名的例子。如果需要访问的 Java 类方法或变量的名与 Scala 类的保留字相同，我们需要使用反引号命名。如：``java.net.Proxy.`type`()``。

- 模式匹配标示符

  在模式匹配表达式中（请回顾 1.4 节中 actor 的相关示例），以小写字母开头的标记会被解析为变量标示符，而大写字母开头的标记则会被解析为常量标示符（如类名）。模式匹配使用了非常简洁的变量表示法，例如无需使用 `val` 关键字，增加该限定能够避免产生某些歧义。

## 语法糖

所有的操作符都是方法。假如你知道该知识点，便能够更容易的理解 Scala 代码。有时候你会看到一些新的操作符，不过你无需担心那些特殊的情况（这些新的操作符本质都是方法，所以无需担心）。1.4 节中出现的 actor 会互相发送异步消息，发送消息时使用了感叹

号！操作符，这一操作符只是一个普通方法罢了。

这种灵活的命名方式让编写出的类库非常自然，就像 Scala 自身的一种延伸。利用这种命名方式，你可以编写一个新的数学库，其中的数值类型支持所有的数学操作。你也可以编写一个与 actor 行为类似的全新的并发消息处理层。除了少数的一些命名规则限制之外，使用这些命名方式能创造出无限的可能。

能够创建操作符符号并不意味着你应该这样做。当你定义 API 时，要提醒自己用户很难读懂这些隐晦的标点符号式的操作符，更别提学会和记住了。滥用这些操作符只会使你的代码变得晦涩。因此，如果你沉迷于创建新的操作符，一旦该操作符无法带来便利，那就意味着你凭空牺牲了方法命名的可读性。

## 3.2 无参数方法

对于那些不包含参数的方法而言，除了可以选择使用中缀调用或后缀调用方式之外，Scala 还允许用户灵活决定是否使用括号。

我们在定义无参方法时可以省略括号。一旦定义无参方法时省略了括号，那么在调用这些方法时必须省略括号。与之相反，假如在定义无参方法时添加了空括号，那么调用方可以选择省略或是保留括号。

例如，`List.size` 的方法定义体中省略了括号，因此你应该编写 `List(1,2,3).size` 这样的代码。假如你尝试输入 `List(1,2,3).size()`，系统将会返回错误。

`java.lang.String` 的 `length` 方法定义体中则包含了括号（这是为了能在 Java 中运行），而 Scala 同时支持 `"hello".length()` 和 `"hello".length` 这两种写法。这同样适用于 Scala 定义的一些定义体中包含空括号的那些无参方法。

为了实现与 Java 语言的互操作，无参方法定义体中出现了是否包含空括号这两种情况的处理规则之间的不一致性。尽管 Scala 也希望定义和使用保持一致（如果定义体中包含括号，那么调用时必须添加括号，反之，调用时必须省略括号），不过由于包含了空括号的定义会更灵活些，这确保了调用 Java 无参方法时可以与调用 Scala 无参方法保持一致。

Scala 社区已经养成了这样一个习惯：定义那些无副作用的无参方法时省略括号，例如：集合的 `size` 方法。定义具有副作用的方法时则添加括号，这样便能提醒读者某些对象可能会发生变化，需要额外小心。假如运行 `scala` 或 `scalac` 时添加了 `-Xlint` 选项，那么在定义那些会产生副作用（例如，方法中会有 I/O 操作）的无参方法时，省略括号将会出现一条警告信息。我在 SBT 编译环境中已经添加了这个标志。

为什么我们会优先讲解是否选择括号的问题呢？这是因为合理考虑是否使用括号有助于构建更具表现力的方法调用链，如下所示，示例中的代码看上去就像是一目了然的“句子”：

```
// src/main/scala/progscala2/rounding/no-dot-better.sc

def isEven(n: Int) = (n % s) == 0

List(1, 2, 3, 4) filter isEven foreach println
```

程序输出如下：

```
2
4
```

尽管上面的语句非常“整齐”，但如果你现在还不熟悉该语句的语法，你就需要花些时间才能理解它的作用。下面四行代码中，每一行都比上一行要少一些，而最后一行便是之前展示的代码。

```
List(1, 2, 3, 4).filter((i: Int) => isEven(i)).foreach((i: Int) => println(i))
List(1, 2, 3, 4).filter(i => isEven(i)).foreach(i => println(i))
List(1, 2, 3, 4).filter(isEven).foreach(println)
List(1, 2, 3, 4) filter isEven foreach println
```

前三行代码更为清晰，因此也更容易为初学者理解。过滤器不过是作用于集合之上的单参数方法，foreach 方法默默地对集合执行了一次循环操作，一旦你熟悉了这些，你会发现最后一行代码，这种“斯巴达式”（斯巴达式，指的是非常简洁的方式）的实现体更易阅读和理解。等你变得更有经验之后，你会认为中间的两行代码就像是妨碍阅读的一种噪音。希望你阅读 Scala 代码的时候能将此牢记于心。

需要澄清的是，由于上面的每个方法都接收单一参数，因此该表达式能正常运行。假如方法链中某一方法接收 0 个或大于 1 个的参数，编译器会困惑。如果出现了这种情况，请为部分或全部方法补上点号。

## 3.3 优先级规则

对于像 `2.0 * 4.0 / 3.0 * 5.0` 这种包含了一系列 Double 操作的表达式，将遵守何种操作符优先级规则呢？下面将按从低到高的顺序列出优先级规则：

1. 所有字母
2. |
3. ^
4. &
5. < >
6. = !
7. :
8. + -
9. * / %
10. 其他特殊字符

同一行的字符具有相同的优先级。不过有一个例外，当 = 用于赋值操作时，该符号的优先级最低。

因为 * 和 / 的优先级相同，因此下面的两段 scala 会话将返回相同值。

```
scala> 2.0 * 4.0 / 3.0 * 5.0
res0: Double = 13.333333333333332

scala> (((2.0 * 4.0) / 3.0) * 5.0)
res1: Double = 13.333333333333332
```

执行由左结合方法组成的方法序列时，只要简单地按照从左到右的顺序执行就行了。那是不是所有的方法都是“左结合”呢？ 答案是否定的。在 Scala 中，任何名字以冒号(:)结尾的方法都与右边的对象所绑定，其他方法则是左绑定的。例如，你可以通过 :: 方法将某一元素放置到列表前面，这一操作成为 cons 操作，cons 是 constructor 的缩写，这也是 Lisp 所引入的概念。

```
scala> val list = List('b', 'c', 'd')
list: List[Char] = List(b, c, d)

scala> 'a' :: list
res4: List[Char] = List(a, b, c, d)
```

第二句表达式等价于 list.::('a')。

任何名字以 : 结尾的方法均与其右边的对象绑定，它们并不与左侧对象绑定。

# 3.4 领域特定语言

领域特定语言，也称为 DSL，指的是为某一专门问题域编写的语言，引入 DSL 是为了方便用简洁直观的方式表达该领域的概念。例如，SQL 便可以被视为一门 DSL，因为它是一门专门用于解释关系模型的编程语言。

不过通常 DSL 只用于即席查询语言，它们要么被嵌入到某一宿主语言内，要么会专门有一个定制的解析器负责解析。嵌入意味着你需要在宿主语言中通过一种方言来实现 DSL。嵌入式 DSL 通常也被称为内部 DSL，而需要特制解析器的 DSL 则被称为外部 DSL。

使用内部 DSL 时，开发者能利用宿主语言的所有特性处理 DSL 未能覆盖的一些临界情况(悲观地说，DSL 是一类存在漏洞的抽象)。内部 DSL 同样免去了编写此法解析器、解析器和其他编写特定语言所需工具的工作。

Scala 为这两类 DSL 均提供了完美的支持。Scala 提供了灵活的标志符规则，如允许使用操作符命名，支持中缀和后缀方法调用语法，这为编写嵌入式 DSL 提供了构建 DSL 所需的组成元素。

## 行为驱动开发

下面的示例应用 ScalaTest 类库（http://www.scalatest.org/），向我们展现了一种被称为行为驱动开发（Behavior-Driven Development）的测试编写方式。Specs2 类库（http://etorreborre.github.io/specs2/）也提供了同样的功能。

```
// src/main/scala/progscala2/rounding/scalatest.scX
// Example fragment of a ScalaTest. Doesn't run standalone.

import org.scalatest.{ FunSpec, ShouldMatchers }
```

```
class NerdFinderSpec extends FunSpec with ShouldMatchers {

  describe ("nerd finder") {
    it ("identify nerds from a List") {
      var actors = List("Rick Moranis", "James Deam", "Woody Allen")
      var finder = new NerdFinder(actors)
      finder.findNerds shouldEqual List("Rick Moranis", "Woody Allen")
    }
  }
}
```

上述示例讲解了如何利用 Scala 编写 DSL，在第 20 章我们会看到更多这样的例子并学会如何动手编写 DSL。

## 3.5 Scala中的if语句

从表面上看，Scala 的 if 语句看起来很像 Java 中的 if 语句。执行 if 语句时先对 if 条件表达式进行估值。假如表达式结果为 true，那么将执行对应的代码块。反之，将测试下一条件分支，以此类推。下面列举了一个简单示例：

```
// src/main/scala/progscala2/rounding/if.sc

if (2 + 2 == 5) {
  println("Hello from 1984.")
} else if (2 + 2 == 3) {
    println("Hello from Remedial Math class?")
} else {
  println("Hello from a non-Orwellian future.")
}
```

Scala 与 Java 语言不同，Scala 中的 if 语句和几乎所有的其他语句都是具有返回值的表达式。因此我们能像下面展示的代码那样，将 if 表达式的结果值赋给其他变量。

```
// src/main/scala/progscala2/rounding/assigned-if.sc

val configFile = new java.io.File("somefile.txt")

val configFilePath = if (configFile.exists()) {
  configFile.getAbsolutePath()
} else {
  configFile.createNewFile()
  configFile.getAbsolutePath()
}
```

if 语句返回值的类型也被称为所有条件分支的最小上界类型，也就是与每条 each 子句可能返回值类型最接近的父类型。在上面这个例子中，configFilePath 是 if 表达式的结果值，该 if 表达式将执行文件不存在的条件分支，并返回新创建文件的绝对路径。将 if 语句的返回值赋予变量 configFilePath 之后，整个应用程序都可以使用该值，其类型为 String 类型。

Scala 中的 if 语句是一类表示式，像 predicate ? trueHandler() : falseHandler() 这种三元表达式对于 Scala 来说是多余的，因此 Scala 并不支持三元表达式。

## 3.6 Scala中的for推导式

除了 if 语句之外，Scala 也为 for 循环这一常见的控制结构提供了非常多的特性，这些 for 循环的特性被称为 for 推导式（for comprehension）或 for 表达式（for expression）。

事实上，推导式一词起源于函数式编程。它表达了这样一个理念：我们遍历一个或多个集合，对集合中的元素进行“推导”，并从中计算出新的事物，新推导出的事物往往是另一个集合。

### 3.6.1 for循环

让我们从一个基本的 for 表达式开始：

```
// src/main/scala/progscala2/rounding/basic-for.sc

val dogBreeds = List("Doberman", "Yorkshire Terrier", "Dachshund",
                     "Scottish Terrier", "Great Dane", "Portuguese Water Dog")

for (breed <- dogBreeds)
  println(breed)
```

你可能已经猜到了，这段代码的意思是：“基于列表 dogBreeds 中的每一个元素，创建临时变量 breed，breed 的值与元素值相同，之后打印 breed。”代码输出如下：

```
Doberman
Yorkshire Terrier
Dachshund
Scottish Terrier
Great Dane
Portuguese Water Dog
```

这种形式不返回任何值，因此它只会执行一些会带来副作用的操作。这类 for 推导式有时候也被称为 for 循环，这与 Java 中的 for 循环较为类似。

### 3.6.2 生成器表达式

像 breed <- dogBreeds 这样的表达式也被称为*生成器表达式*（generator expression），生成器表达式之所以叫这个名字，是因为该表达式会基于集合生成单独的数值。左箭头操作符 (<-) 用于对像列表这样的集合进行遍历。

我们还可以使用生成器表达式对某些区间进行访问，以这种方式编写出的 for 循环更加自然。

```
// src/main/scala/progscala2/rounding/generator.sc

for (i <- 1 to 10) println(i)
```

### 3.6.3 保护式：筛选元素

怎样才能获得更细的操作粒度呢？ 我们可以加入 if 表达式，来筛选出我们希望保留的元

素。这些表达式也被称为*保护式*（guard）。为了能够从犬种列表中挑选中㹴犬，我们对之前的代码进行了修改，具体如下：

```
// src/main/scala/progscala2/rounding/guard-for.sc

val dogBreeds = List("Doberman", "Yorkshire Terrier", "Dachshund",
                     "Scottish Terrier", "Great Dane", "Portuguese Water Dog")
for (breed <- dogBreeds
  if breed.contains("Terrier")
) println(breed)
```

输出如下：

```
Yorkshire Terrier
Scottish Terrier
```

你还可以在 `for` 循环中添加多个保护式：

```
// src/main/scala/progscala2/rounding/double-guard-for.sc

val dogBreeds = List("Doberman", "Yorkshire Terrier", "Dachshund",
                     "Scottish Terrier", "Great Dane", "Portuguese Water Dog")

for (breed <- dogBreeds
  if breed.contains("Terrier")
  if !breed.startsWith("Yorkshire")
) println(breed)

for (breed <- dogBreeds
  if breed.contains("Terrier") && !breed.startsWith("Yorkshire")
) println(breed)
```

在第二个 `for` 推导式中，两个 `if` 语句被合并为一个语句。这两个 `for` 推导式的输出如下所示：

```
Scottish Terrier
Scottish Terrier
```

### 3.6.4 Yielding

假如你并不需要打印过滤后的集合，你需要编写代码对过滤后的集合进行处理，那么该怎么办呢？使用 `yield` 关键字便能在 `for` 表达式中生成新的集合。

另外，我们将转而使用大括号代替圆括号，以相似的方法把参数列表封装在大括号中时可以使得块结构的格式看起来更为直观：

```
// src/main/scala/progscala2/rounding/yielding-for.sc

val dogBreeds = List("Doberman", "Yorkshire Terrier", "Dachshund",
                     "Scottish Terrier", "Great Dane", "Portuguese Water Dog")
val filteredBreeds = for {
  breed <- dogBreeds
  if breed.contains("Terrier") && !breed.startsWith("Yorkshire")
} yield breed
```

每次执行 for 表达式时，过滤后的结果将生成 breed 值。随着代码的执行，这些结果值逐渐积累起来，累计而成的结果值集合被赋给了 filteredBreeds 对象。for-yield 表达式所生成的集合类型将根据被遍历的集合类型推导而出。在上面的例子中，由于 filteredBreeds 源于 dogBreeds 列表，而 dogBreeds 类型为 List[String]，因此 filteredBreeds 的类型为 List[String]。

for 推导式有一个不成文的约定：当 for 推导式仅包含单一表达式时使用原括号，当其包含多个表达式时使用大括号。值得注意的是，使用原括号时，早前版本的 Scala 要求表达式之间必须使用分号。

假如一个 for 推导式并未使用 yield，而是执行像打印这样的具有副作用的操作，那么我们将其称为 for 循环。这是因为它的行为更像是你所熟悉的 Java 和其他语言中的 for 循环。

### 3.6.5 扩展作用域与值定义

Scala 的 for 推导式还有一个有用的特征：你能够在 for 表达式中的最初部分定义值，并可以在后面的表达式中使用该值。如下所示：

```
// src/main/scala/progscala2/rounding/scoped-for.sc

val dogBreeds = List("Doberman", "Yorkshire Terrier", "Dachshund",
                     "Scottish Terrier", "Great Dane", "Portuguese Water Dog")
for {
  breed <- dogBreeds
  upcasedBreed = breed.toUpperCase()
} println(upcasedBreed)
```

需要注意的是，尽管 upcasedBreed 的值不可变，但并不需要使用 val 关键字进行限定[2]执行结果如下：

```
DOBERMAN
YORKSHIRE TERRIER
DACHSHUND
SCOTTISH TERRIER
GREAT DANE
PORTUGUESE WATER DOG
```

如果你想到了 Option，那就可以用在这个示例中。正如我们之前讨论的那样，Option 是 null 更好的替代方案，Option 是一类特殊形式的集合，它只包含 0 个或 1 个元素，意识到这一点对你会有帮助。我们也可以理解下面代码：

```
// src/main/scala/progscala2/patternmatching/scoped-option-for.sc

val dogBreeds = List(Some("Doberman"), None, Some("Yorkshire Terrier"),
                     Some("Dachshund"), None, Some("Scottish Terrier"),
                     None, Some("Great Dane"), Some("Portuguese Water Dog"))
```

注 2：在 Scala 早先的版本中，val 关键字是可选的，而现在则不推荐使用该关键字。

```
println("first pass:")
for {
  breedOption <- dogBreeds
  breed <- breedOption
  upcasedBreed = breed.toUpperCase()
} println(upcasedBreed)

println("second pass:")
for {
  Some(breed) <- dogBreeds
  upcasedBreed = breed.toUpperCase()
} println(upcasedBreed)
```

想象我们会调用返回各种犬种名称的一些服务。这些服务会返回 Option 类型，而由于一些服务无法返回任何值，这些服务将返回 None。在第一个 for 推导式的第一个表达式中，每一个被提取的元素均为 Option 对象。而后续的代码行中将使用箭头符提取 option 中的值。

稍等！当你尝试从 None 对象中提取对象时难道不会抛出异常吗？的确如此，不过由于此时推导式会进行有效地检查并忽略 None，因此不会有异常抛出。这正如我们在第二行代码之前增加了显式的 if breedOption != None。

第二个 for 推导式使用了模式匹配，这使得代码更为清新。只有当 BreedOption 是 Some 类型时，表达式 Some(breed) <- dogBreeds 才会成功执行并提取出 breed，所有操作一步完成。None 元素不再被处理。

什么时候使用左箭头（<-），什么时候该使用等于号（=）呢？当你遍历某一集合或其他像 Option 这样的容器并试图提取值时，你应该使用箭头。当你执行并不需要迭代的赋值操作时，你应使用等于号。for 推导式的第一句表达式必须使用箭头符执行抽取 / 迭代操作。

在大多数语言的循环体中，你可以使用跳出循环、也可以继续进行迭代。Scala 并未提供 break 和 continue 语句，不过编写地道的 Scala 代码时，你几乎不需要使用这些语句。你可以使用条件表达式或者使用递归判断循环是否应该继续。如果你能在一开始便对集合进行过滤以消除循环中的复杂条件，那就更好了[3]。

> Scala 的 for 推导式并不提供 break 和 continue 功能。Scala 提供的其他特性使得这两个功能没有存在的必要。

## 3.7 其他循环结构

Scala 提供了许多其他的循环结构，由于 for 推导式是如此的灵活和强大，这些结果并未得到广泛的应用。另外，有时候你需要的仅仅是一个 while 循环。

注 3：不过，考虑到存在对 break 功能的需求，Scala 提供了 scala.util.control.Breaks 对象（http://www.scala-lang.org/api/current/#scala.util.control.Breaks$），该对象可用于实现 break 功能，不过我从未用过该功能，你最好也不用它。

### 3.7.1 Scala的while循环

只要判断条件成立，while 循环将一直运行对应代码块。例如，在下一个 13 号的周五到来之前，下面的代码将按照一天一次的频率打印抱怨的信息：

```
// src/main/scala/progscala2/rounding/while.sc
// 警告:这个脚本会运行非常非常长时间!
import java.util.Calendar

def isFridayThirteen(cal: Calendar): Boolean = {
  val dayOfWeek = cal.get(Calendar.DAY_OF_WEEK)
  val dayOfMonth = cal.get(Calendar.DAY_OF_MONTH)

  // Scala将最后一个表达式的结果值作为该方法的返回结果
  (dayOfWeek == Calendar.FRIDAY) && (dayOfMonth == 13)
}

while (!isFridayThirteen(Calendar.getInstance())) {
  println("Today isn't Friday the 13th. Lame.")
  // sleep for a day
  Thread.sleep(86400000)
}
```

### 3.7.2 Scala中的do-while循环

与 while 循环相似，只要条件表达式返回 true，do-while 循环语句就会执行代码。也就是说，执行完代码块后，do-while 语句便会检查条件是否为真。为了能计数十次，我们可以编写如下代码：

```
// src/main/scala/progscala2/rounding/do-while.sc

var count = 0

do {
  count += 1
  println(count)
} while (count < 10)
```

## 3.8 条件操作符

Scala 从 Java 及其祖先处继承了大多数的条件操作符。你能在 if 语句、while 循环以及每个应用了 else 条件的地方看到表 3-1 所列的那些条件操作符。

表3-1：条件操作符

| 操作符 | 操　作 | 描　述 |
|---|---|---|
| && | 和操作 | 操作符左边和右边的值都为 true。只有当操作符左边的值为真值时才会评估右边值是否为真 |
| \|\| | 或操作 | 操作符左边和右边值至少有一个为 true。只有当操作符左边值为 false 时才会评估右边值是否为真 |

（续）

| 操作符 | 操　作 | 描　述 |
|---|---|---|
| > | 大于 | 操作符左边的值应大于右边的值 |
| >= | 大于或等于 | 操作符左边的值大于或等于右边的值 |
| < | 小于 | 操作符左边的值应小于右边的值 |
| <= | 小于或等于 | 操作符左边的值小于或等于右边的值 |
| == | 等于 | 操作符左边的值等于右边的值 |
| != | 不等于 | 操作符左边的值不等于右边的值 |

需要注意的是 `&&` 和 `||` 是*短路*（short-circuiting）操作符，一旦得知结果，这些操作便会停止对表达式估值。

绝大多数操作符的行为与它们在 Java 和其他语言中的表现一致，但操作符 `==` 和它的逆操作符 `!==` 例外。在 Java 中，`==` 只会对对象引用进行比较，它并不会执行一次逻辑上的相等检查，即比较字段值。而你调用 `equals` 方法便是出于这个目的。因此假如有两个类型相同且具有相同字段值（也就是说，这两个对象的状态相同）的不同对象，在 Java 中执行 `==` 操作将返回 false。

与之相反，Scala 使用 `==` 符执行逻辑意义上的相等检查，不过该操作符也调用了 `equals` 方法。假如你并不希望进行逻辑相等检查（我们会在 10.6 节详细讨论对象相等的相关内容），而只想比较引用，你可以使用 Scala 提供的新的方法 `eq`。

## 3.9　使用try、catch和final子句

Scala 推崇通过使用函数式结构和强类型以减少对异常和异常处理的依赖的编码范式。尽管如此，异常仍然有用，而当 Scala 需要与普遍使用异常的 Java 代码交互时，异常尤为重要。

> 与 Java 不同，Scala 并不支持已被视为失败设计的检查型异常（checked exception）。Scala 将 Java 中的检查型异常视为非检查型，而且方法声明中也不包含 `throw` 子句。不过 Scala 提供了有助于 Java 互操作的 `@throws` 注解（http://www.scala-lang.org/api/current/index.html#scala.throws），具体内容请参考 23.2 节。

Scala 将异常处理作为另一类模式匹配来进行处理，因此我们可以简洁地对各种不同类型的异常进行处理。

让我们看看 Scala 在资源管理这样一个常见的应用场景中是如何处理异常的。我们希望通过某种方式打开并处理一些文件。在这个示例中我们仅仅会统计行数。不过，我们仍然必须对一些错误场景进行处理。比如说，文件也许并不存在，这个错误尤其是当我们需要让用户指定文件名时尤为明显。除此之外，处理文件时可能也会有某些错误（为了测试错误发生的场景，我们将随意地触发一个错误）。无论是否成功地对文件进行了处理，我们都需要确保关闭了所有的文件句柄。

```scala
// src/main/scala/progscala2/rounding/TryCatch.scala
package progscala2.rounding

object TryCatch {
  /** Usage: scala rounding.TryCatch filename1 filename2 ... */
  def main(args: Array[String]) = {
    args foreach (arg => countLines(arg))                        // ❶
  }

  import scala.io.Source                                         // ❷
  import scala.util.control.NonFatal

  def countLines(fileName: String) = {                           // ❸
    println()  // Add a blank line for legibility
    var source: Option[Source] = None                            // ❹
    try {                                                        // ❺
      source = Some(Source.fromFile(fileName))                   // ❻
      val size = source.get.getLines.size
      println(s"file $fileName has $size lines")
    } catch {
      case NonFatal(ex) => println(s"Non fatal exception! $ex")     // ❼
    } finally {
      for (s <- source) {                                        // ❽
        println(s"Closing $fileName...")
        s.close
      }
    }
  }
}
```

❶ 使用 foreach 循环遍历参数列表并对各个参数进行处理。该循环每遍历一次便返回一个 Unit 对象，而 foreach 执行完毕后所返回的最终结果也是 Unit 对象。

❷ 导入用于读取输入的 scala.io.Source 类（http://www.scala-lang.org/api/current/#scala.io.Source）以及用于匹配 nonfatal 异常的 scala.util.control.NonFatal 类（http://www.scala-lang.org/api/current/#scala.util.control.NonFatal$）。

❸ 统计每个文件名所对应文件的行数。

❹ 由于我们将变量 source 声明为 Option 类型，因此我们在 finally 子句中能分辨出 source 对象是否是真正的实例。

❺ 开始执行 try 子句。

❻ 假如文件不存在，source.fromFile 方法将抛出 java.io.FileNotFoundException（http://docs.oracle.com/javase/8/docs/api/java/io/FileNotFoundException.html）类型的异常。否则的话，我们将该方法的返回值封装到 Some 对象中。下一行中我们将调用 source 变量的 get 方法，由于目前我们已经能确认 source 属于 Some 类型，因此这一操作是安全的。

❼ 捕获那些非致命的错误。例如，内存不足是一个致命错误。

❽ 使用 for 推导式从 Some 类型的对象中提取 Source 实例，之后将关闭文件。假如 source 对象为 None，将不会发生任何事。

请留意 catch 子句。Scala 会使用模式匹配来捕捉你所希望捕获的异常，而 Java 则使用单

独的 catch 子句来捕获各个异常。与 Java 相比，Scala 捕获异常的方式更紧凑、更灵活。在这段示例代码中，case NonFatal(ex) => …子句使用 scala.util.control.NonFatal 匹配了所有的非致命性异常。

应用 finally 子句可以确保资源会在一处得到合理的清理。如果不使用 finally，我们将不得不分别在 try 子句和 catch 子句中重复清理逻辑，这样才能确保文件句柄会被关闭。由于我们使用了 for 推导式从 option 对象中抽取出 Source 对象，因此即使 option 对象实际上是一个 None 实例，什么也不会发生，包含了文件 close 方法的代码块并不会被调用。

> 假如你需要对 Option 对象进行检测，当它是 Some 对象时执行一些操作，而当它是 None 对象时则不进行任何操作，那么你可以使用 for 推导式，这也是 Scala 的一个广泛应用的常见用法。

由于该程序已经经过 sbt 编译，因此我们可以在 sbt 提示符后运行 run-main 任务来启动该程序，启动 run-main 任务时允许输入参数。为了方便阅读，我将输入参数折成多行，使用 \ 表示行符 ，并删除了一些文本。

```
> run-main progscala2.rounding.TryCatch foo/bar \
  src/main/scala/progscala2/rounding/TryCatch.scala
[info] Running rounding.TryCatch foo/bar .../rounding/TryCatch.scala

... java.io.FileNotFoundException: foo/bar (No such file or directory)

file src/main/scala/progscala2/rounding/TryCatch.scala has 30 lines
Closing src/main/scala/progscala2/rounding/TryCatch.scala...
[success] ...
```

第一个文件 foo/bar 并不存在，而第二个文件才是该程序的源文件。使用 scala.io.Source API 能够方便地对来自文件或其他源的数据流进行处理。与一些这类的 API 相似，文件不存在时 Source 将抛出异常。因此读取 foo/bar 文件时抛出异常的行为是可预期的行为。

> 假如无论是否成功地使用了资源，资源都需要被清理，请将资源清理的相关逻辑放到 finally 子句中执行。

除了使用模式匹配定位异常之外，Scala 异常处理的其他设定与大多数语言相似。与 Java 一样，使用 Scala 时我们通过编写 throw new MyBadExceptoin(...) 抛出异常。假如你自定义的异常是一个 case 类，那么抛出异常时可以省略 new 关键字。这就是 Scala 与其他语言在异常处理上的差别。

自动资源管理是一类常见的 Scala 设计模式。为了实现自动资源管理，Joshua Suereth 编写了一个名为 ScalaARM（http://jsuereth.com/scala-arm/）的独立项目。下面让我们尝试编写自动资源管理程序。

## 3.10 名字调用和值调用

我们曾动手实现了可重用资源管理器，其具体实现如下：

```
// src/main/scala/progscala2/rounding/TryCatchArm.scala
package progscala2.rounding
import scala.language.reflectiveCalls
import scala.util.control.NonFatal

object manage {
  def apply[R <: { def close():Unit }, T](resource: => R)(f: R => T) = {
    var res: Option[R] = None
    try {
      res = Some(resource)         // 只会引用"resource"一次!!
      f(res.get)
    } catch {
      case NonFatal(ex) => println(s"Non fatal exception! $ex")
    } finally {
      if (res != None) {
        println(s"Closing resource...")
        res.get.close
      }
    }
  }
}

object TryCatchARM {
  /** Usage: scala rounding.TryCatch filename1 filename2 ... */
  def main(args: Array[String]) = {
    args foreach (arg => countLines(arg))
  }

  import scala.io.Source

  def countLines(fileName: String) = {
    println()  // 打印空白行,以增加可读性
    manage(Source.fromFile(fileName)) { source =>
      val size = source.getLines.size
      println(s"file $fileName has $size lines")
      if (size > 20) throw new RuntimeException("Big file!")
    }
  }
}
```

你只要将命令行中的 `TryCatch` 替换为 `TryCatchArm`，便可以像运行之前的示例那样运行该程序并得到相似的输出。

能够将关注点分离（separation of concern，SOC）固然很好，不过为了实现这点，我们需要运用一些强大的新工具。

首先，我们将对象命名为 `manage`，而不是 `Manage`。通常我们都遵循类型名称首字母大写的规范，不过由于该示例使用 `manage` 的方式与使用函数的方式相似，因此未遵循该规范。我们希望 `manage` 在用户代码中看上去像一个内置的操作符，整个代码看起来就像是一个

while 循环。我们可以在 countLines 方法中查看 manage 对象的使用方式。这个示例也演示了如何使用 Scala 工具构建小型领域特定语言（DSL）。

## manage.apply方法

manage.apply 方法声明看上去非常奇怪，为了能够理解该声明，我们将对其进行分解。下面再次列出了该方法的签名，我们将分行显示方法签名，而每一行也都提供了对应的注释。

```
def apply[
  R <: { def close():Unit },    ❶
  T ]                           ❷
  (resource: => R)              ❸
  (f: R => T) = {...}           ❹
```

❶ 这行出现了两个新的事物。R 表示我们将要管理的资源类型。而 <: 则意味着 R 属于某其他类型的子类。在本例中 R 的父类型是一个包含 close():Unit 方法的结构类型。为了能帮助你更直观的理解 R 类型，尤其是当你之前没接触过结构化类型时，你可以认为 R <: Closable 表示 Closable 接口中定义了 close():Unit 方法并且 R 实现了 Closable 接口。不过结构化类型允许我们使用反射机制嵌入包含 close():Unit 方法的任意类型（如 Source 类型）。反射机制会造成许多系统开销，而结构化类型代价也较为昂贵，因此就像后缀表达式那样，Scala 将反射列为可选特性，为了能够让编译器相信我们知道我们在做什么，需要在代码中添加 import 语句。

❷ 我们传入了用于处理资源的匿名函数，而 T 表示该匿名函数的返回值。

❸ 尽管看上去像是一个声明体较为奇特的函数，但 resource 事实上是一个传名参数（by-name parameter）。我们暂且将其视为一个在调用时应省略括号的函数。

❹ 最后我们将传入第二个参数列表，其中包含了一个输入为 resource、返回值类型为 T 的匿名函数，该匿名函数将负责处理 resource。

我们再回到注释 1，假设所有资源均实现了 Closable 抽象，那么 apply 方法声明看起来会是下面这个样子：

```
object manage {
  def apply[ R <: Closable, T](resource: => R)(f: R => T) = {...}
  ...
}
```

resource 只会在 val res = Some(resource) 这行代码中被求值，这行代码必不可少的。由于 resource 的表现与函数相似，因此就像是一个会被重复调用的函数，每次引用该变量时便会对其求值。但我们并不希望每次引用 resource 时都会执行一次 Source.fromFile(fileName)，因为这意味着我们每次都会重新打开一个新的 Source 实例。

之后，我们将 res 值传入工作函数 f 中。

TryCatchARM.countLines 又是如何使用 manage 对象的呢？ manage 对象在这段代码中看上去就像是一个 Scala 自带的控制结构，该控制结构包含了两个参数列表：一个用于创建 Source 对象，而另一个则是处理 Source 对象的代码块。这样一来，manage 对象看起来就像是一个普通 while 语句了。

我们再回顾一下之前的代码，创建 Source 对象的第一条表达式其实并没有立刻执行，在进入 manage 对象之前，该表达式都没有被执行。直到执行 manage 对象内的代码 val res = Some(resource) 时，该表达式才会执行。这便是传名参数 resource 能提供的功能。我们编写的 manage.apply 方法可以接受任意表达式输入，但这些表达式将延后执行。

与大多数语言相似，Scala 通常使用*按值调用*（call-by-value）的语法。如果 manage(Source.fromFile(fileName)) 所处上下文遵循按值调用的方式的话，那么 Scala 将执行 Source.fromFile 方法，并将返回值传递给 manage 对象。

通过将 Source.fromFile 推迟到 apply 方法中的代码行 val res = Some(resource)，该行代码等效于下列代码：

```
val res = Some(Source.fromFile(fileName))
```

正是因为要支持像延迟计算这样的语法，Scala 才提供了传名参数。

假如 Scala 未提供传名参数，该怎么办呢？我们可以使用匿名函数实现延迟计算，不过这种实现方式看起来略显丑陋。

对 manage 对象的调用代码看上去像是下列代码：

```
manage(() => Source.fromFile(fileName)) { Source =>
```

而在 apply 方法体中，将以函数调用的方式来引用 resource。

```
val res = Some(resource())
```

尽管这种函数调用并不会给我们造成可怕的后果，不过像 manage 对象那样，我们也可以通过*按名调用*（call by name）构建自己的控制结构。

请记住，传名参数的行为与函数相似；每次使用该参数时便会执行表达式。在我们的 ARM 示例里，我们希望该表达式只会被执行一次，但这并不能反映通常的情况。

下面的示例中通过定义 continue 结构，实现了一个简单的类似于 while 循环的结构体：

```
// src/main/scala/progscala2/rounding/call-by-name.sc

@annotation.tailrec                                                // ❶
def continue(conditional: => Boolean)(body: => Unit) {             // ❷
  if (conditional) {                                               // ❸
    body                                                           // ❹
    continue(conditional)(body)
  }
}

var count = 0                                                      // ❺
continue(count < 5) {
  println(s"at $count")
  count += 1
}
```

❶ 确保了调用实现体时将采用尾递归的方式。

❷ 定义 continue 函数，该函数接受两个参数列表：第一个列表中仅包含了一个传值参数

conditional，而第二个列表则包含了传值参数 body。body 代表了每次迭代都会执行的代码体。

❸ 检查当前是否满足条件。

❹ 假如满足条件，将执行 body 参数，并递归调用 `continue` 函数。

❺ 调用 `continue` 方法！

读者需谨记一点：传名参数会在每次被引用时估值。（顺便提一下，上述实现描述了如何使用递归取代循环结构。）由于传名参数的求值会被推迟，并可能会一再地被重复调用，因此此类参数具有惰性。除此之外，Scala 也提供了*惰性值*（lazy value）。

## 3.11 惰性赋值

惰性赋值是与传名参数相关的技术，假如你希望以延迟的方式初始化某值，并且表达式不会被重复计算，则需要使用惰性赋值。下面列举了一些需要用到该技术的常见场景。

- 由于表达式执行代价昂贵（例如：打开一个数据库连接），因此我们希望能推迟该操作，直到我们确实需要表达式结果值时才执行它。
- 为了缩短模块的启动时间，可以将当前不需要的某些工作推迟执行。
- 为了确保对象中其他的字段的初始化过程能优先执行，需要将某些字段惰性化。我们会在 11.4 节讨论 Scala 对象模型时深入探讨这些场景。

下面的示例中便应用了惰性赋值：

```
// src/main/scala/progscala2/rounding/lazy-init-val.sc

object ExpensiveResource {
  lazy val resource: Int = init()
  def init(): Int = {
    // 执行某些代价高昂的操作
    0
  }
}
```

`lazy` 关键字意味着求值过程将会被推迟，只有需要时才会执行计算。

那么惰性赋值与方法调用有那些差别呢？对于方法调用而言，每次调用方法时方法体都会被执行；而惰性赋值则不然，首次使用该值时，用于初始化的“代码体”才会被执行一次。这种只能执行一次的计算对于可变字段而言几乎没有任何意义。因此，`lazy` 关键字并不能用于修饰 `var` 变量。

我们通过*保护式*（guard）来实现惰性值。当客户代码引用了惰性值时，保护式会拦截引用并检查此时是否需要初始化惰性。由于保护式能确保惰性值在第一次访问之前便已初始化，因此增加保护式检查只有当第一次引用惰性值时才是必要的。但不幸的是，很难解除之后的惰性值保护式检查。所以，与“立刻”值相比，惰性值具有额外的开销。因此只有当保护式带来的额外开销小于初始化带来的开销时，或者将某些值惰性化（请参考 11.4 节）能简化系统初始化过程并确保执行顺序满足依赖条件时，你才应该使用惰性值。

## 3.12 枚举

还记得之前那个列出各类犬种的示例吗？我们也许期望能有一个顶级的 `Breed` 类型来纪录各类犬种。像 `Breed` 这样的类型被统称为枚举类型，而该类型包含的值被称为枚举值。

虽然许多编程语言都内置了枚举值，但是 Scala 却使用了另外一种实现方式：在标准库中专门定义 `Enumeration` 类（http://www.scala-lang.org/api/current/scala/Enumeration.html）。这意味着 Scala 并未提供任何特殊语法来支持枚举，而 Java 则不然。所以 Scala 语言中的枚举与 Java 中的 `enum` 结构在字节码层面上没有任何联系。

请看示例：

```
// src/main/scala/progscala2/rounding/enumeration.sc

object Breed extends Enumeration {
  type Breed = Value
  val doberman = Value("Doberman Pinscher")
  val yorkie   = Value("Yorkshire Terrier")
  val scottie  = Value("Scottish Terrier")
  val dane     = Value("Great Dane")
  val portie   = Value("Portuguese Water Dog")
}
import Breed._

// 打印所有犬种及其ID列表
println("ID\tBreed")
for (breed <- Breed.values) println(s"${breed.id}\t$breed")

// 打印㹴犬列表
println("\nJust Terriers:")
Breed.values filter (_.toString.endsWith("Terrier")) foreach println

def isTerrier(b: Breed) = b.toString.endsWith("Terrier")

println("\nTerriers Again??")
Breed.values filter isTerrier foreach println
```

该程序会打印以下信息：

```
ID      Breed
0       Doberman Pinscher
1       Yorkshire Terrier
2       Scottish Terrier
3       Great Dane
4       Portuguese Water Dog

Just Terriers:
Yorkshire Terrier
Scottish Terrier

Terriers Again??
Yorkshire Terrier
Scottish Terrier
```

我们会发现犬种枚举类型中包含了许多 `Value` 类型值，如下所示：

```
val doberman = Value("Doberman Pinscher")
```

实际上，每个犬种声明均调用了接收单一字符串参数的 `Value` 方法。我们使用该方法为每个枚举值指定了较长的犬种名称。调用 `Value.toString` 方法将返回该字符串。

`Breed` 类型是一个别名，无需使用各个 `Value` 值，仅用 `Breed` 枚举便能定位到具体犬种。只有在输入 `isTerrie` 方法参数时我们才真正需要使用 `Value` 值。假如我们注释 `Breed` 的类型定义，那么该函数就无法通过编译。

尽管类型名和方法名均为 `Value`，但它们之间并不存在命名空间冲突。因为编译器为值和方法分别维护了各自独立的命名空间。

Scala 还提供了一些其他的重载版 `Value` 方法。我们之前使用的 `Value` 方法接收单一字符串输入，而另一个 `Value` 方法则不接受任何输入参数。无参的 `Value` 方法将对象名作为输入字符串，例如：变量 `doberman` 对应的字符串是 doberman。第三个 `Value` 方法的输入参数是一个整型 `ID` 值，该方法在使用默认字符串（即变量名）的同时会将我们显式指定的整数值作为 `ID` 值。最后一个 `Value` 方法同时接收整数和字符串输入。

由于我们调用的方法并未显式指定 ID 值，`Value` 对象的 ID 将会自动从 0 开始分配，并按照声明的顺序逐一递增。这些 `Value` 方法都会生成 `Value` 对象，而这些新创建的 `Value` 对象也会被添加到枚举的 `Value` 集合中。

通过调用了 `value` 方法，我们能像处理集合那样处理这些枚举值。这样一来，我们便能使用 `for` 推导式遍历所有的犬种，对这些犬种按名称进行过滤，也能查看系统为那些未指定 ID 的犬种自动分配的 ID 值。

就像这个示例表现的那样，我们通常希望能给枚举值取一个可读性强的名字。但是有时候你也许又不需要对枚举值命名，下面这一示例改编自 Scaladoc 文档中枚举类型（http://www.scala-lang.org/api/current/index.html#scala.Enumeration）的入口页的示例。

```
// src/main/scala/progscala2/rounding/days-enumeration.sc

object WeekDay extends Enumeration {
  type WeekDay = Value
  val Mon, Tue, Wed, Thu, Fri, Sat, Sun = Value
}
import WeekDay._

def isWorkingDay(d: WeekDay) = ! (d == Sat || d == Sun)

WeekDay.values filter isWorkingDay foreach println
```

运行上述脚本将输出下列信息（v2.7）：

```
Mon
Tue
Wed
Thu
Fri
```

请注意，我们引入了 WeekDay._，这使得每一个枚举值（如 Mon、Tues）都在代码的作用域内。否则的话，你需要编写像 WeekDay.Mon、WeekDay.Tus 这样的代码。我们同样可以通过调用 values 方法遍历枚举值。在这个示例中，我们就过滤出“工作日”对应的枚举值。

尤其与 Java 相比，枚举在 Scala 代码中出现的次数并不多。尽管 Scala 中的枚举使用便利，但是需要预先知道集合中应包含哪些枚举值。而客户是无法增加其他的枚举值的。

作为枚举的一种替代品，case 类常常应用于那些需要使用“枚举值”的场景中。尽管 case 类更重量级一些，但是却具有两大优势。首先，case 类允许添加方法和字段，而且我们也能够对枚举值应用模式匹配，这便为用户提供了更好的灵活性。其次 case 类能适用于包含未知枚举值的场景。只要有需要，用户代码便可以将更多的 case 类添加到基本集合中。

## 3.13 可插入字符串

我们在 1.4 节已经介绍了什么是可插入字符串（interpolated string），这里将对其进行更深层次地分析。

假如某一字符串的形式为 s"foo ${bar}"，那么表达式 bar 将会被转化为字符串并被插入到原字符串中的 $"{bar}" 的位置中。假如表达式 bar 返回的不是字符串，而是某一类型的实例，那么只要该实例中定义了 toString 方法，系统便会调用该方法将该实例转化成字符串。如果 bar 表达式返回值无法转换成字符串，那么程序将会报错。

如果 bar 是一个变量引用，那么可以省略字符串中的大括号。如下所示：

```
val name = "Buck Trends"
println(s"Hello, $name")
```

如果想在可插入字符串中输入美元符，那么请连续输入两个美元符 $$。

Scala 中存在两类可插入字符串。第一类采用 printf 格式，这类可插入字符串以 f 为前缀。第二类也被称为“原生的”可插入字符串，这类字符串并不会对像 \n 这样的逃逸字符串进行扩展。

假设我们正在生成财务报表，希望浮点数只显示到小数点后两位[4]。那么可以编写如下代码：

```
val gross   = 100000F
val net     = 64000F
val percent = (net / gross) * 100
println(f"$$${gross}%.2f vs. $$${net}%.2f or ${percent}%.1f%%")
```

最后一行代码将输出下列结果：

```
 $100000.00 vs. $64000.00 or 64.0%
```

采用 printf 格式时，Scala 会调用 Java 的 Formatter 类。这类字符串中嵌入的表达式与之前代码使用的语法相同，均为 $"{...}"，不过在编写时 printf 格式指令与 ${...} 之间不

---

注 4：通常并不使用原生的浮点数和双精度浮点数表示货币，这是因为表示货币金额时需要满足各种记账规则（如舍入规则）。不过为了简化起见，我们在示例中使用了浮点数。

应有空格。

在该示例中，为了打印出一个美元符号，我们使用了两个美元符 $$；同时使用了两个百分号 %% 以打印出一个百分号。表达式 ${gross}%.2f 会格式化 gross 的变量值，保留其小数点后两位数字。

可插入字符串中的变量类型必须与其格式吻合，为此 Scala 会执行一些隐式转化。下面列出的代码试图在浮点数语境中使用 Int 表达式，Scala 允许这种操作并会在整数的小数点后添加两个 0。而第二个表达式则尝试将 Double 类型值展现为 Int 类型，不过这次尝试会导致编译错误：

```
scala> val i = 200
i: Int = 200

scala> f"${i}%.2f"
res4: String = 200.00

scala> val d = 100.22
d: Double = 100.22

scala> f"${d}%2d"
<console>:9: error: type mismatch;
 found   : Double
 required: Int
              f"${d}%2d"
                  ^
```

顺便提一下，调用 Java 静态方法 String.format（http://docs.oracle.com/javase/8/docs/api/java/lang/String.html）将按照 printf 的格式对字符串格式化。该方法的输入参数包含了格式字符串以及一组用于替代格式字符串的变量列表。String.format 方法还有另外一个版本，该版本中的第一个参数代表字符串的区域设置（该参数为 Locale 类型）。

Scala 编译器会在某些语境中对 Java 字符串进行封装并提供一些额外的方法，这些定义在 scala.collection.immutable.StringLike（http://www.scala-lang.org/api/current/scala/collection/immutable/StringLike.html）中。这些额外提供的方法中包含了一个叫作 format 的实例方法。你可以对一个格式化字符串调用该方法并传入需要插入到该字符串中的参数列表。具体如下：

```
scala> val s = "%02d: name = %s".format(5, "Dean Wampler")
s: String = "05: name = Dean Wampler"
```

在该示例中，我们希望能输出两位数整数并在小数点添加两个零。最后一类内置的字符串插入器被称为“原生”（raw）插入器，该插入器不会对控制字符进行扩展，具体请参阅下面两个示例。

```
scala> val name = "Dean Wampler"
name: String = "Dean Wampler"

scala> s"123\n$name\n456"
res0: String =
123
```

```
Dean Wampler
456

scala> raw"123\n$name\n456"
res1: String = 123\nDean Wampler\n456
```

最后提一下，其实我们可以自定义字符串插入器，不过在此之前，我们需要掌握更多的关于隐式转换（implicit）的知识。如果想了解具体细节，请参阅 5.3.1 节。

## 3.14 Trait：Scala语言的接口和“混入”

尽管我们已经学习了快 100 页的内容，但我们至今仍未讨论到面向对象语言的一个最基础的特性：在 Scala 中如何定义抽象？ Java 语言中的接口是抽象的同义词，那 Scala 呢？Scala 又该如何实现类继承呢？

Scala 从函数式编程思想中汲取了 trait，为了突出它的强大能力，我有意将讲解相关内容的篇幅推后，不过现在是时候对这一重要主题进行概览了。

我曾使用过一些模糊的术语，比如说抽象。一些示例也用抽象类来表示父类。但我之前并未思索过这些用词，只是认为曾经在其他语言中见过类似的结构体。

Java 提供了接口，你可以在接口中声明方法，但却不能定义方法。至少在 Java 8 诞生之前是这样的。同样，你也可以在接口中声明和定义静态变量或者嵌套类型。

Scala 使用 trait 来替代接口。在第 9 章我们会详细地讲述 trait，目前你可以将其视为允许将声明方法实现的接口。使用 trait 时，你可以声明示例字段（与 Java 接口一样，你在 trait 中并非只局限于声明静态字段）并选择是否定义这些字段，你也可以参照之前的枚举示例，在 trait 中声明或定义类型。

trait 提供的这些扩展被证实确实能够打破 Java 对象模型的一些局限。Java 对象模型只允许在类中定义方法和字段，而 trait 则允许真正意义上的组合行为（“混入”模式）。这在 Java 8 诞生之前是很难实现的。

下面示例解决了每一个企业级 Java 开发者都曾面对的问题：在应用中混入日志。我们首先编写了下列服务：

```
// src/main/scala/progscala2/rounding/traits.sc

class ServiceImportante(name: String) {
  def work(i: Int): Int = {
    println(s"ServiceImportante: Doing important work! $i")
    i + 1
  }
}

val service1 = new ServiceImportante("uno")
(1 to 3) foreach (i => println(s"Result: ${service1.work(i)}"))
```

该服务执行了某些工作并返回下列输出：

```
ServiceImportante: Doing important work! 1
Result: 2
ServiceImportante: Doing important work! 2
Result: 3
ServiceImportante: Doing important work! 3
Result: 4
```

现在我们在该服务中混入一个标准日志库，为了简单起见，我们将使用 `println` 输出日志。

示例中包含了两个 trait，其中的一个定义了日志抽象（不包含任何具体的成员），而另一个则实现了日志抽象，将日志信息输出到标准输出。

```
trait Logging {
  def info   (message: String): Unit
  def warning(message: String): Unit
  def error  (message: String): Unit
}

trait StdoutLogging extends Logging {
  def info   (message: String) = println(s"INFO:    $message")
  def warning(message: String) = println(s"WARNING: $message")
  def error  (message: String) = println(s"ERROR:   $message")
}
```

请注意，这里编写的日志与 Java 中的接口非常相似。它们的 JVM 字节码甚至采用了相同的实现方式。

最后，我们声明了一个“混入”了日志功能的服务：

```
val service2 = new ServiceImportante("dos") with StdoutLogging {
  override def work(i: Int): Int = {
    info(s"Starting work: i = $i")
    val result = super.work(i)
    info(s"Ending work: i = $i, result = $result")
    result
  }
}
(1 to 3) foreach (i => println(s"Result: ${service2.work(i)}"))

INFO:    Starting work: i = 1
ServiceImportante: Doing important work! 1
INFO:    Ending work: i = 1, result = 2
Result: 2
INFO:    Starting work: i = 2
ServiceImportante: Doing important work! 2
INFO:    Ending work: i = 2, result = 3
Result: 3
INFO:    Starting work: i = 3
ServiceImportante: Doing important work! 3
INFO:    Ending work: i = 3, result = 4
Result: 4
```

现在服务开启或结束工作时都会输出日志信息。

为了能混入 trait，我们需要使用 `with` 关键字。根据自身需要，我们可以混入任意多个

trait。一些 trait 也许不会对现有行为做任何修改，它们只会添加一些新的有用方法，但这些方法之间是相互独立的。

实际上，为了能注入日志，在该示例中我们修改了服务的行为，但并未修改服务与客户之间的“契约”，也就是说，我们并未修改服务的对外行为。[5]

假如我们希望能在 ServiceImportante 的多个实例中混入 StdoutLogging 特征，我们可以声明一个新类：

```
class LoggedServiceImportante(name: String)
  extends ServiceImportante(name) with stdoutLogging {...}
```

请留意我们是如何将参数 name 传递给父类 ServiceImportante 的。在创建实例时，new LoggedServiceImportante("tres") 能够实现你想要的功能。

不过，假如我们仅需将日志特征混入一个实例，我们可以在定义变量时混入特征。

为了使用日志扩展，我们不得不对 work 方法进行覆写。假如你希望覆盖父类中某一具体方法，那么 Scala 要求必须输入 override 关键字。请留意我们是如何访问父类的 work 方法的。与 Java 和一些其他语言一样，我们通过 super.work 调用该方法。

trait 和对象组合还有很多值得讨论的地方，本书会在后续章节讲述相关的内容。

## 3.15　本章回顾与下一章提要

前三章讲述了许多基础知识，从中我们了解到 Scala 代码可以如此的简洁灵活。经过本章的学习，我们掌握了一些强大的可用于定制 DSL 或操作数据的结构，比如说 for 推导式。最后学习了如何使用枚举封装值以及 trait 的一些基本知识。

现在你应该已经具备了阅读一些 Scala 代码的能力，不过 Scala 语言还有很多需要学习的地方。现在我们将开启 Scala 特征的深度之旅。

注 5：严格意义上说，这种说法并不正确。这个附加的 I/O 操作已经对代码与外界之间的交互产生了影响。

# 第 4 章
# 模式匹配

乍一看，*模式匹配*似乎与你喜欢的类 C 语言中的 case 语句很相似。因为在常见的类 C 语言 case 语句中，你只能按顺序匹配简单的数据类型和表达式。例如，“在 i 为 5 的情况下，打印一条消息；在 i 为 6 的情况下，退出程序”。

而在 Scala 的模式匹配中，可以使用类型、通配符、序列、正则表达式，甚至可以深入获取对象的状态。这种对象状态的获取遵循一定的协议，也就是对象内部状态的可见性由该类型的实现来控制，这使得我们能够轻易获取暴露的状态并应用于变量中。对象状态的获取往往被称为“提取”或“解构”。

模式匹配可以用在许多代码场景中。最常用于 match 语句中，之后我们会给出其他的用法。在 1.4 节的 actor 示例代码中，我们介绍了两个相对简单的 match 语句实例。在 2.4 节，我们也进一步讨论了其他用法。

## 4.1 简单匹配

首先，我们通过匹配 Boolean 值来模拟掷硬币：

```
// src/main/scala/progscala2/patternmatching/match-boolean.sc

val bools = Seq(true, false)

for (bool <- bools) {
  bool match {
    case true => println("Got heads")
    case false => println("Got tails")
  }
}
```

这看起来就像是 C 风格的 case 语句。为了试验一下，你可以尝试将第二个 case  false 语句注释掉，再运行脚本。这时你会得到一个警告和一个错误消息：

```
<console>:12: warning: match may not be exhaustive.
It would fail on the following input: false
                bool match {
                ^
Got heads
scala.MatchError: false (of class java.lang.Boolean)
  at .<init>(<console>:11)
  at .<clinit>(<console>)
  ...
```

由于序列类型存在两种可能的取值：true 或 false，因此编译器警告 match 语句未能覆盖所有可能的输入值。当尝试去匹配一个没有 case 语句的值时，我们发现编译器抛出了 MatchError（http://www.scala-lang.org/api/current/scala/MatchError.html）。

我得提一下，以上例子的另一种替代写法是旧式的 if 语句：

```
for (bool <- bools) {
  val which = if (bool) "head" else "tails"
  println("Got " + which)
}
```

## 4.2 match中的值、变量和类型

接下来，我们来讨论几种 match 语句。以下的例子能匹配特定的某个值，也能匹配特定类型的所有值，同时展示了 default 语句的写法来匹配任意输入值。

```
// src/main/scala/progscala2/patternmatching/match-variable.sc

for {
  x <- Seq(1, 2, 2.7, "one", "two", 'four)                       // ❶
} {
  val str = x match {                                            // ❷
    case 1          => "int 1"                                   // ❸
    case i: Int     => "other int: "+i                           // ❹
    case d: Double  => "a double: "+x                            // ❺
    case "one"      => "string one"                              // ❻
    case s: String  => "other string: "+s                        // ❼
    case unexpected => "unexpected value: " + unexpected         // ❽
  }
  println(str)                                                   // ❾
}
```

❶ 由于序列元素包含不同类型，因此序列的类型为 Seq[Any]。

❷ x 的类型为 Any。

❸ 如果 x 等于 1 则匹配。

❹ 匹配除 1 外的其他任意整数值。将 x 的值安全地转为 Int，并赋值给 i。

❺ 匹配所有 Double 类型，x 的值被赋给 Double 型变量 d。

❻ 匹配字符串“one”。

❼ 匹配除“one”外的其他任意字符串，x 的值被赋给了 String 类型的变量 x。

❽ 匹配其他任意输入，x 的值被赋给 unexpected 这个变量。由于未给出任何类型说明，unexpected 的类型被推断为 Any，起到了 default 语句的功能。

❾ 打印返回的字符串。

为了使代码直观一些，我将 =>(“箭头”) 排成一列。以下是程序的输出：

```
int 1
other int: 2
a double 2.7
string one
other string: two
unexpected value: 'four
```

像所有表达式一样，match 语句也会返回一个值。在这里，所有的子句都返回字符串，因此整个子句的返回值类型为 String。编译器会推断所有 case 子句返回值类型的*最近公共父类型*（也称为最小公共上限）作为返回值类型。

由于 x 类型为 Any，因此我们需要足够的子句来覆盖所有可能的输入值。（对比一下我们用来匹配 Boolean 值的第一个例子。）这就是我们需要“default 子句”（使用 unexpected）的原因。然而，编写偏函数时，我们不需要覆盖所有可能的类型，因为它们是被有意设计的。

匹配是按顺序进行的，因此具体的子句应该出现在宽泛的子句之前。否则，具体的语句将不可能有机会被匹配上。所以，默认子句必须是最后一个子句。幸运的是，编译器能识别这种类型的错误。

由于舍入误差的存在，两个看似相等的值可能由于最后一位有效数字的不同而被判断为不相等，我没有在示例中使用匹配浮点数字面量的子句。

以下是对之前例子的简单变形：

```
// src/main/scala/progscala2/patternmatching/match-variable2.sc

for {
  x <- Seq(1, 2, 2.7, "one", "two", 'four)
} {
  val str = x match {
    case 1          => "int 1"
    case _: Int     => "other int: "+x
    case _: Double  => "a double: "+x
    case "one"      => "string one"
    case _: String  => "other string: "+x
    case _          => "unexpected value: " + x
  }
  println(str)
}
```

我们用占位符 _ 替换了变量 i、d、s 和 unexpected。事实上我们并不需要这些类型的对应变量值，只需要产生字符串。所以，可以在所有子句中使用 x。

除了偏函数，所有的 match 语句都必须是完全覆盖所有输入的。当输入类型为 Any 时，在结尾用 case _ 或 case some_name 作为默认子句。

编写 case 子句时，有一些规则和陷阱需要注意。在被匹配或提取的值中，编译器假定以大写字母开头的为类型名，以小写字母开头的为变量名。

以下示例中的这条规则可能会使你感到惊讶：

```
// src/main/scala/progscala2/patternmatching/match-surprise.sc

def checkY(y: Int) = {
  for {
    x <- Seq(99, 100, 101)
  } {
    val str = x match {
      case y => "found y!"
      case i: Int => "int: "+i
    }
    println(str)
  }
}

checkY(100)
```

在第一个 case 子句中，我们希望它能匹配上一个可以由我们来指定的值，而不是一个硬编码的值。所以，我们可能会希望第一个 case 子句在 x 等于 y 时成功匹配，y 的值为 100。脚本执行时将产生以下输出：

```
int: 99
found y!
int: 101
```

以下是我们获得的实际输出：

```
<console>:12: warning: patterns after a variable pattern cannot match (SLS 8.1.1)
If you intended to match against parameter y of method checkY, you must use
backticks, like: case `y` =>
             case y => "found y!"
                  ^
<console>:13: warning: unreachable code due to variable pattern 'y' on line 12
             case i: Int => "int: "+i
                                   ^
<console>:13: warning: unreachable code
             case i: Int => "int: "+i
                                   ^
checkY: (y: Int)Unit
found y!
found y!
found y!
```

case  y 的含义其实是：匹配所有输入（由于这里没有类型注解），并将其赋值给新的变量 y。

这里的 y 没有被解释为方法参数 y。因此，事实上我们将一个默认的、匹配一切的语句写在了第一个，导致系统给出了这条“变量型匹配语句”会匹配一切输入的警告。我们的代码也从未执行到第二条 case 语句，于是就得到了两条关于不可达代码的警告。在这里 SLS 8.1.1 指《Scala 语法规范》（http://www.scala-lang.org/docu/files/ScalaReference.pdf）的 8.1.1 节。

第一条错误信息已经告诉我们应该怎么做：使用反引号表示真正想要匹配的是参数 y 的值。

```
// src/main/scala/progscala2/patternmatching/match-surprise-fix.sc

def checkY(y: Int) = {
  for {
    x <- Seq(99, 100, 101)
  } {
    val str = x match {
      case `y` => "found y!"            // 只修改了这一行：`y`
      case i: Int => "int: "+i
    }
    println(str)
  }
}
checkY(100)
```

这时，输出就符合我们的期望了。

在 case 子句中，以小写字母开头的标识符被认为是用来提取待匹配值的新变量。如果需要引用之前已经定义的变量时，应使用反引号将其包围。与此相对，以大写字母开头的标识符被认为是类型名称。

有时不同的匹配子句需要使用相同的处理代码，此时，为了避免代码冗余，我们可以将相同处理代码重构为一个单独的方法。同时，case 子句也持“或”逻辑，使用 | 方法即可：

```
// src/main/scala/progscala2/patternmatching/match-variable2.sc

for {
  x <- Seq(1, 2, 2.7, "one", "two", 'four)
} {
  val str = x match {
    case _: Int | _: Double => "a number: "+x
    case "one"              => "string one"
    case _: String          => "other string: "+x
    case _                  => "unexpected value: " + x
  }
  println(str)
}
```

现在，Int 和 Double 类型的值都能匹配上第一个 case 子句了。

## 4.3　序列的匹配

Seq（http://www.scala-lang.org/api/current/scala/collection/Seq.html，表示“序列”）是具体的

集合类型的父类型，这些集合类型支持以确定顺序遍历其元素，如 List（http://www.scala-lang.org/api/current/scala/collection/immutable/List.html）和 Vector（http://www.scala-lang.org/api/current/scala/collection/immutable/Vector.html）。

我们来考察用模式匹配和递归方法遍历 Seq 的传统方法，顺便学习一些关于序列的基础知识。

```
// src/main/scala/progscala2/patternmatching/match-seq.sc

val nonEmptySeq    = Seq(1, 2, 3, 4, 5)                          // ❶
val emptySeq       = Seq.empty[Int]
val nonEmptyList   = List(1, 2, 3, 4, 5)                         // ❷
val emptyList      = Nil
val nonEmptyVector = Vector(1, 2, 3, 4, 5)                       // ❸
val emptyVector    = Vector.empty[Int]
val nonEmptyMap    = Map("one" -> 1, "two" -> 2, "three" -> 3)   // ❹
val emptyMap       = Map.empty[String,Int]

def seqToString[T](seq: Seq[T]): String = seq match {           // ❺
  case head +: tail => s"$head +: " + seqToString(tail)          // ❻
  case Nil => "Nil"                                              // ❼
}

for (seq <- Seq(                                                 // ❽
    nonEmptySeq, emptySeq, nonEmptyList, emptyList,
    nonEmptyVector, emptyVector, nonEmptyMap.toSeq, emptyMap.toSeq)) {
  println(seqToString(seq))
}
```

❶ 构造一个非空的 Seq[Int]（事实上返回了一个 List）；然后用惯用方法构造了一个空的 Seq[Int]。

❷ 构造一个非空的 List[Int]（Seq 的一个子类型）；然后用 Scala 库的一个专用对象 Nil（http://www.scala-lang.org/api/current/#scala.collection.immutable.Nil$），表示任意类型的空 List。

❸ 构造一个非空的 Vector[Int]（http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.Vector）（Seq 的一个子类型）；然后构造了一个空的 Vector[Int]。

❹ 构造了一个非空的 Map[String,Int]（http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.Map），这不是 Seq 的子类型。在接下来的讨论中我们会涉及这一点。Map[String,Int] 的键为 String 类型，值为 Int 类型。然后构造了一个空的 Map[String,Int]。

❺ 定义了一个递归方法，从 Seq[T] 中构造 String，T 为某种待定的类型。方法体是用来与输入的 Seq[T] 相匹配。

❻ 这里存在两个互斥的 match 子句。第一个子句匹配非空的 Seq，提取其头部（第一个元素）以及尾部（除头部以外，剩下的元素）。（Seq 有 head 和 tail 方法，但在这里，这两个标识符按 case 子句的惯例被解释为变量。）在 case 子句中，用提取的头部加上“+:”，以及尾部的字符串表示来构造一个字符串。尾部的字符串表示由调用 seqToString 产生。

❼ 另外一个 case 只可能是空 Seq。我们用表示空 List 专用的对象 Nil 来匹配。注意，任何 Seq 的尾部都可以认为是以一个相应类型的空 Seq，事实上，List 就是这么实现的。

❽ 将以上这些 Seq 作为元素放到另一个大 Seq 中（对其中的 Map 调用 toSeq，将其转为键 – 值对组成的序列），然后遍历该序列，并打印各个序列调用 seqToString 返回的结果。

以下为执行结果：

```
1 +: 2 +: 3 +: 4 +: 5 +: Nil
Nil
1 +: 2 +: 3 +: 4 +: 5 +: Nil
Nil
1 +: 2 +: 3 +: 4 +: 5 +: Nil
Nil
(one,1) +: (two,2) +: (three,3) +: Nil
Nil
```

Map 并不是 Seq 的子类型，因为 Map 不保证遍历的顺序是固定的。因此，我们调用 Map.toSeq 创建一个键 – 值元组的序列。由此得到的 Seq 键值对（pair）将会按照插入的顺序进行遍历。这种副作用仅限于小的 Map 的实现上，并不是针对所有 Map 的一般性保证。这里的空集合表明 seqToString 方法对空集也能正常工作。

这里有两种新的 case 子句。第一个子句中，head +: tail 匹配了序列的头部和尾部。+: 操作符是序列的“构造”操作符，与我们在优先规则中为 List 使用的 :: 操作符类似。回想一下，以冒号 (:) 结尾的方法向右结合，即向 Seq 的尾部结合。

虽然它们被称作“操作符”和“方法”，但其实并不太准确；我们会在之后的章节讨论这些表达式，现在先来关注几个关键点。

首先，case 子句只匹配至少包含一个头部元素的非空序列，它将序列的头部和剩下的部分分别提取到可变变量 head 和 tail 中。

第二，重申一下，这里的 head 和 tail 是任意的两个变量名。然而，Seq 类型也分别存在两个名为 head 和 tail 的方法，用于提取序列的头部和尾部元素。通常情况下，我们可以从上下文中清晰地看出这两个标识符是函数还是变量。顺便提一下，对空序列调用这两个方法时，编译器会抛出异常。

Seq 的行为很符合链表的定义，因为在链表中，每个头结点除了含有自身的值以外，还指向链表的尾部（即链表剩下的元素），从而创建了一种层级结构，类似以下四个节点所组成的的序列。在这里尾部添加了一个空序列：

```
(node1, (node2, (node3, (node4, (end)))))
```

Scala 库中有一个名为 Nil 的对象，可以匹配所有的空序列。我们甚至可以用 Nil 来表示非 List 的其他空集合，因为序列对相等操作的实现都是一样的，不必精确匹配具体类型。

以下是上述程序的一个变体，增加了括号，这次我们只使用了几个集合类型：

```
// src/main/scala/progscala2/patternmatching/match-seq-parens.sc

val nonEmptySeq    = Seq(1, 2, 3, 4, 5)
val emptySeq       = Seq.empty[Int]
```

```
val nonEmptyMap    = Map("one" -> 1, "two" -> 2, "three" -> 3)

def seqToString2[T](seq: Seq[T]): String = seq match {
  case head +: tail => s"($head +: ${seqToString2(tail)})"          // ❶
  case Nil => "(Nil)"
}

for (seq <- Seq(nonEmptySeq, emptySeq, nonEmptyMap.toSeq)) {
  println(seqToString2(seq))
}
```

❶ 重新格式化字符串，增加了外边的括号，(…)。

如下所示，脚本输出清楚地显示了层级结构，每个“子列表”都被括号包围：

```
(1 +: (2 +: (3 +: (4 +: (5 +: (Nil))))))
(Nil)
((one,1) +: ((two,2) +: ((three,3) +: (Nil))))
```

我们只用了两个 `case` 子句和递归就处理了序列。这暗示了所有序列的基础特性：序列要么为空，要么非空。这听起来很老套，但一旦理解了这一点，你就会多一个基于“分治”法的工具。`processSeq` 就多次使用该方法。

在 Scala 2.10 之前，处理 `List` 有另一种很相似的惯用方法：

```
// src/main/scala/progscala2/patternmatching/match-list.sc

val nonEmptyList = List(1, 2, 3, 4, 5)
val emptyList    = Nil

def listToString[T](list: List[T]): String = list match {
  case head :: tail => s"($head :: ${listToString(tail)})"          // ❶
  case Nil => "(Nil)"
}

for (l <- List(nonEmptyList, emptyList)) { println(listToString(l)) }
```

❶ 用 `::` 代替 `+:`。

输出也很类似：

```
(1 :: (2 :: (3 :: (4 :: (5 :: (Nil))))))
(Nil)
```

因此，使用 `Seq` 写代码更方便，因为它可以应用于所有子类型，包括 `List` 与 `Vector`。

我们可以通过复制、粘贴上一个示例的输出，重新构造出原始的对象：

```
scala> val s1 = (1 +: (2 +: (3 +: (4 +: (5 +: Nil)))))
s1: List[Int] = List(1, 2, 3, 4, 5)

scala> val l  = (1 :: (2 :: (3 :: (4 :: (5 :: Nil)))))
l: List[Int] = List(1, 2, 3, 4, 5)

scala> val s2 = (("one",1) +: (("two",2) +: (("three",3) +: Nil)))
s2: List[(String, Int)] = List((one,1), (two,2), (three,3), (four,4))
```

```
scala> val m  = Map(s2 :_*)
m: scala.collection.immutable.Map[String,Int] =
  Map(one -> 1, two -> 2, three -> 3, four -> 4)
```

值得注意，Map.apply 工厂方法需要一个可变参数列表，其中的参数是由 2 个元素组成的元组。所以，为了用序列 s2 来构造 Map，我们只能用 :_* 惯用法来将序列 s2 转化为可变参数列表。

使用 +: 和 :: 时，构造方法与模式匹配（“解构”）有着优雅的对称关系。我们将在本章的后面部分探究它的实现方式，并提供可分析的示例。

# 4.4 元组的匹配

通过元组字面量，很容易对元组进行匹配：

```
// src/main/scala/progscala2/patternmatching/match-tuple.sc

val langs = Seq(
  ("Scala",   "Martin", "Odersky"),
  ("Clojure", "Rich",   "Hickey"),
  ("Lisp",    "John",   "McCarthy"))

for (tuple <- langs) {
  tuple match {
    case ("Scala", _, _) => println("Found Scala")                  // ❶
    case (lang, first, last) =>                                     // ❷
      println(s"Found other language: $lang ($first, $last)")
  }
}
```

❶ 匹配一个三元组的元组，其中第一个元素为字符串 Scala，忽略第二和第三个元素。

❷ 匹配任意三元素元组，其中的元素可以为任意类型，但在这里，由于输入的值为 lang，元素类型被推断为 String。元组的三个元素被提取到变量 lang、first 和 last 中。

该示例的输出如下：

```
Found Scala
Found other language: Clojure (Rich, Hickey)
Found other language: Lisp (John, McCarthy)
```

元素可以拆分为各个组成的元素。我们可以匹配元组中任意位置的字面量，同时忽略我们不需要的值。

# 4.5 case中的guard语句

在模式匹配中使用字面量的好处很多，但有时你却需要添加其他的逻辑：

```
// src/main/scala/progscala2/patternmatching/match-guard.sc
```

```
for (i <- Seq(1,2,3,4)) {
  i match {
    case _ if i%2 == 0 => println(s"even: $i")                    // ❶
    case _             => println(s"odd:  $i")                    // ❷
  }
}
```

❶ 只有当 i 为偶数时才匹配。我们用 _ 代替变量名，因为我们已经有 i 可以作为变量名了。

❷ 匹配上一 case 子句相对的另一种可能性，即 i 为奇数。

以上代码的输出为：

```
odd:  1
even: 2
odd:  3
even: 4
```

注意，if 表达式的两边不需要使用括号，就像我们在 for 表达式中也不需要括号一样。

## 4.6 case类的匹配

我们现在来看更多关于深度匹配的例子，并对 case 类对象的内容进行考察：

```
// src/main/scala/progscala2/patternmatching/match-deep.sc

// Simplistic address type. Using all strings is questionable, too.
case class Address(street: String, city: String, country: String)
case class Person(name: String, age: Int, address: Address)

val alice   = Person("Alice",   25, Address("1 Scala Lane", "Chicago", "USA"))
val bob     = Person("Bob",     29, Address("2 Java Ave.",  "Miami",   "USA"))
val charlie = Person("Charlie", 32, Address("3 Python Ct.", "Boston",  "USA"))

for (person <- Seq(alice, bob, charlie)) {
  person match {
    case Person("Alice", 25, Address(_, "Chicago", _) => println("Hi Alice!")
    case Person("Bob", 29, Address("2 Java Ave.", "Miami", "USA")) =>
      println("Hi Bob!")
    case Person(name, age, _) =>
      println(s"Who are you, $age year-old person named $name?")
  }
}
```

输出如下：

```
Hi Alice!
Hi Bob!
Who are you, 32 year-old person named Charlie?
```

我们可以匹配嵌套类型的内容。以下的例子使用的元组更接近真实生活。想象一下，我们有一个 (String,Double) 元组组成的序列，表示商店里的商品名称与商品价格，同时想要将它们连同序号一起打印出来。为解决上面的问题，我们可以用到 Seq.zipWithIndex 方法：

```
// src/main/scala/progscala2/patternmatching/match-deep-tuple.sc

val itemsCosts = Seq(("Pencil", 0.52), ("Paper", 1.35), ("Notebook", 2.43))
val itemsCostsIndices = itemsCosts.zipWithIndex
for (itemCostIndex <- itemsCostsIndices) {
  itemCostIndex match {
    case ((item, cost), index) => println(s"$index: $item costs $cost each")
  }
}
```

我们在 REPL 里运行以上脚本，用 :load 命令观察变量的类型和运行的输出（略加整理了一下格式）：

```
scala> :load src/main/scala/progscala2/patternmatching/match-deep-tuple.sc
Loading src/main/scala/progscala2/patternmatching/match-deep-tuple.sc...
itemsCosts: Seq[(String, Double)] =
  List((Pencil,0.52), (Paper,1.35), (Notebook,2.43))
itemsCostsIndices: Seq[((String, Double), Int)] =
  List(((Pencil,0.52),0), ((Paper,1.35),1), ((Notebook,2.43),2))
0: Pencil costs 0.52 each
1: Paper costs 1.35 each
2: Notebook costs 2.43 each
```

调用 zipWithIndex 时，返回的元组形式为 ((name,cost),index)。通过匹配这种形式，我们提取了输入元组的三个元素，并将其打印出来。这样的代码是我经常使用的。

## 4.6.1 unapply方法

除了 Scala 库里的类型以外，我们自定义的 case 类也可以使用类型匹配与提取的功能，它甚至还支持深度嵌套。

这是如何实现的呢？我们在 1.4 节已经了解到：case 类有一个伴随对象，伴随对象中有一个名为 apply 的工厂方法，用于构造对象。基于"对称"的观点，我们可以推断，一定还存在另一个名为 unapply 的自动生成的方法，用于提取和"解构"。事实上，确实存在这样的一个用于提取的方法，当遇见如下形式的类型匹配表达式时，该方法就会被调用：

```
person match {
  case Person("Alice", 25, Address(_, "Chicago", _)) => ...
  ...
}
```

Scala 找到 Person.unapply(…) 和 Address.unapply(…)，然后调用这两个函数。所有的 unapply 方法都返回 Option[TupleN[…]]，此处的 N 表示可以从对象中提取的值的个数。在 Person 这个 case 类中，N 为 3。被提取的值的类型与相应位置元组元素的类型一致。对于 Person 而言，提取值的类型分别为 String、Int 和 Address。所以，编译器生成的 Person 的伴随对象是这样的：

```
object Person {
  def apply(name: String, age: Int, address: Address) =
    new Person(name, age, address)
  def unapply(p: Person): Option[Tuple3[String,Int,Address]] =
    Some((p.name, p.age, p.address))
```

```
  ...
}
```

既然编译器已经知道对象是 Person，为什么 unapply 的返回值还要用 Option 呢？ Scala 允许 unapply 方法“否决”这个匹配，返回 None，这时，Scala 会使用下一个 case 子句。另外，如果我们不希望的话，可以不必暴露对象的所有属性。例如，如果我们不想暴露年龄隐私的话，可以选择不暴露 age。我们会在 unapplySeq 方法中详细探讨这一细节，不过此时，只要知道被提取的属性以 Some 的形式返回即可，在本例中 Some 中的内容为 Tuple3。编译器随后将元组 Tuple3 中的元素与字面值进行比较，比如匹配字符串 Alice；或者赋值给我们已经命名的变量；亦或者用 _ 占位符丢掉不需要的元素。

> 为了获得性能上的优势，Scala 2.11.1 放松了对 unapply 必须返回 Option[T] 的要求。现在 unapply 能返回任意类型，只要该类型具有以下方法：
>
> ```
> def isEmpty: Boolean
> def get: T
> ```

如果有必要，unapply 方法会被递归调用。像本例中，我们在 Person 中有一个嵌套的 Address 对象。类似地，在元组的那个例子中，我们也相应地递归调用了 unapply 方法。

case 关键字被同时用于声明一种“特殊”的类，又用于 match 表达式中的 case 表达式，这可不是巧合。case 类的特性就是为更便捷地进行模式匹配而设计的。

在继续探索之前，注意一下，返回类型 Option[Tuple3[String,Int,Address]] 的写法显得太过冗长。Scala 允许我们用元组字面语法来处理这种类型：

```
val t1: Option[Tuple3[String,Int,Address]] = ...
val t2: Option[(String,Int,Address)] = ...
val t3: Option[ (String, Int, Address) ] = ...
```

元组字面语法使得代码更易阅读；此外，逗号后的空格也有助于提高代码的可读性。

现在让我们回到神秘的 head +: tail 表达式，去真正理解它的含义。我们可以看到 +:（构造）操作符可以通过在现有序列前追加新元素来构造新序列，我们可以这样凭空开始构造整个序列：

```
val list = 1 +: 2 +: 3 +: 4 +: Nil
```

由于 +: 是一个向右结合的操作符，因此我们首先将 4 追加到 Nil 中，再将 3 追加到产生的序列中，以此类推。

Scala 希望尽可能地支持构造和解构 / 提取的标准语法。这一点我们已经在序列、列表和元组中看到。另外，这些操作都是成对的，互为逆操作。

如果我们可以用一个名为 +: 的方法完成序列的构造，那么用相同的语法形式如何完成解构操作呢？刚才研究了 unapply 方法，但这是正确答案吗？虽然 Person.unapply 和 TupleN.unapply 知道在它们的实例中有多少“东西”，分别是 3 个和 N 个。但现在我们希望支持任意非空的集合。

为了完成这一点，Scala 库定义了一个特殊的单例对象，名为 +:（http://www.scala-lang.org/api/current/index.html#scala.collection.$plus$colon$）。是的，对象的名字为“+:”，类似于方法名，在 Scala 中，类型名可以使用的字符也很广泛。

这个类型只有一个方法，即编译器用来在 case 语句中进行提取操作的 unapply 方法。unapply 方法声明的示意就像是这样（我对实际声明稍微做了些简化，因为我们还没有涉及全部知识细节，不需要很多类型系统的知识来理解完整的方法签名）：

```
def unapply[T, Coll](collection: Coll): Option[(T, Coll)]
```

头部的类型被推断为类型 T；尾部被推断为某种集合类型 Coll。Coll 同时也是输入的集合类型。于是，方法返回了一个 Option，其内容为输入集合的头部和尾部组成的两元素元组。

编译器如何才能在看到 case head +: tail => ... 表达式时调用 +:.unapply(collection) 方法呢？我们需要把 case 子句写成 case +:(head, tail) => ... 才能与刚才示例中的模式匹配保持一致。

事实上，我们可以写成如下形式：

```
scala> def processSeq2[T](l: Seq[T]): Unit = l match {
     |   case +:(head, tail) =>
     |     printf("%s +: ", head)
     |     processSeq2(tail)
     |   case Nil => print("Nil")
     | }

scala> processSeq2(List(1,2,3,4,5))
1 +: 2 +: 3 +: 4 +: 5 +: Nil
```

我们也可以使用中缀表达式 head +: tail，这是编译器的又一个语法糖。包含有两个类型参数的类型可以写为中缀表达式，同样，case 子句也可以这么做。考虑如下 REPL 中的代码：

```
// src/main/scala/progscala2/patternmatching/infix.sc
scala> case class With[A,B](a: A, b: B)
defined class With

scala> val with1: With[String,Int] = With("Foo", 1)
with1: With[String,Int] = With(Foo,1)

scala> val with2: String With Int  = With("Bar", 2)
with2: With[String,Int] = With(Bar,2)

scala> Seq(with1, with2) foreach { w =>
     |   w match {
     |     case s With i => println(s"$s with $i")
     |     case _        => println(s"Unknown: $w")
     |   }
     | }
Foo with 1
Bar with 2
```

所以我们可以用两种形式书写类型签名：With[String,Int] 或者 String With Int。后者更

易阅读，不过缺乏经验的 Scala 程序员可能会感到困惑。请记住，尝试用类似的语法形式初始化一个值是不可行的：

```
// src/main/scala/progscala2/patternmatching/infix.sc
scala> val w = "one" With 2
<console>:7: error: value With is not a member of String
       val w = "one" With 2
                     ^
```

`List` 也有这样的一个类似的对象——`::`（http://www.scala-lang.org/api/current/scala.collection.immutable.$colon$colon）。如果你希望逆序处理序列中的每个元素时该怎么办呢？可以用一个对象进行处理！ Scala 库的对象 `:+`（http://www.scala-lang.org/api/current/scala.collection.$colon$plus$）可以让你匹配 `List` 的最后一个元素，然后从后往前依次访问各元素：

```
// src/main/scala/progscala2/patternmatching/match-reverse-seq.sc
// Compare to match-seq.sc

val nonEmptyList   = List(1, 2, 3, 4, 5)
val nonEmptyVector = Vector(1, 2, 3, 4, 5)
val nonEmptyMap    = Map("one" -> 1, "two" -> 2, "three" -> 3)

def reverseSeqToString[T](l: Seq[T]): String = l match {
  case prefix :+ end => reverseSeqToString(prefix) + s" :+ $end"
  case Nil => "Nil"
}

for (seq <- Seq(nonEmptyList, nonEmptyVector, nonEmptyMap.toSeq)) {
  println(reverseSeqToString(seq))
}
```

输出如下：

```
Nil :+ 1 :+ 2 :+ 3 :+ 4 :+ 5
Nil :+ 1 :+ 2 :+ 3 :+ 4 :+ 5
Nil :+ (one,1) :+ (two,2) :+ (three,3)
```

`Nil` 是第一个输出的元素，它的结合性是左结合的。同时，对于其他两个输入的 `List` 和 `Vector`，也生成了相同的输出。

你应该比较一下 `seqToString` 和 `reverseSeqToString` 方法，因为它们各自使用不同的方式实现递归。在此之前，请确保你理解它们各自的工作机制。

像之前一样，你可以用输出的内容重新构造一个集合（第二行输出与第一行重复，可以忽略）：

```
scala> Nil :+ 1 :+ 2 :+ 3 :+ 4 :+ 5
res0: List[Int] = List(1, 2, 3, 4, 5)
```

对于 `List`，用于追加元素的 :+ 方法以及用于模式匹配的 :+ 方法均需要 $O(n)$ 的时间复杂度，这两个方法都必须要从列表的头部遍历一遍。而对于其他某些序列，如 `Vector`，则需要 $O(1)$ 的时间复杂度。

## 4.6.2 unapplySeq方法

如果你想要更灵活一些，希望提取序列时返回非固定数量的元素，该怎么办呢？unapplySeq 方法可以做到这一点。除了 apply 方法以外，Seq 的伴随对象还实现 unapplySeq 方法，而不是普通伴随对象的 unapply 方法：

```
def apply[A](elems: A*): Seq[A]
def unapplySeq[A](x: Seq[A]): Some[Seq[A]]
```

回顾一下，在这里 A* 表示 elems 是一个可变参数列表。下面的示例是对之前例子中 +: 的变形，这里我们将触发 unapplySeq 方法的调用，并考察所提取元素的“滑动窗口”：

```
// src/main/scala/progscala2/patternmatching/match-seq-unapplySeq.sc

val nonEmptyList   = List(1, 2, 3, 4, 5)                          // ❶
val emptyList      = Nil
val nonEmptyMap    = Map("one" -> 1, "two" -> 2, "three" -> 3)

// Process pairs
def windows[T](seq: Seq[T]): String = seq match {
  case Seq(head1, head2, _*) =>                                  // ❷
    s"($head1, $head2), " + windows(seq.tail)                    // ❸
  case Seq(head, _*) =>
    s"($head, _), " + windows(seq.tail)                          // ❹
  case Nil => "Nil"
}

for (seq <- Seq(nonEmptyList, emptyList, nonEmptyMap.toSeq)) {
  println(windows(seq))
}
```

❶ 定义了一个非空 List，一个 Nil，和一个 Map。

❷ 在 match 语句中，看起来我们似乎会隐含调用 Seq.apply(…)，但实际上我们调用的是 Seq.unapplySeq。我们提取了前两个元素，忽略了用 _* 表示的其他可变参数。* 表示另个或多个元素，与正则表达式中的 * 类似。

❸ 用匹配到的前两个元没素格式化字符串，同时调用 seq.tail 将提取的“窗口”向后移动一位。注意，在这次匹配中，我们并有提取尾部元素。

❹ 我们还需要匹配只有一个元素的序列，否则匹配就不完全。用 _ 表示不存在的“第二个”元素。我们已经知道调用 windows(seq.tail) 会返回 Nil，但为了避免将字符串再重复一遍，我们再次调用了 windows 方法。

滑动窗口的输出为：

```
(1, 2), (2, 3), (3, 4), (4, 5), (5, _), Nil
Nil
((one,1), (two,2)), ((two,2), (three,3)), ((three,3), _), Nil
```

我们依然可以在匹配中使用 +:，这个写法更加优雅：

```
// src/main/scala/progscala2/patternmatching/match-seq-without-unapplySeq.sc
```

```
val nonEmptyList   = List(1, 2, 3, 4, 5)
val emptyList      = Nil
val nonEmptyMap    = Map("one" -> 1, "two" -> 2, "three" -> 3)

// Process pairs
def windows2[T](seq: Seq[T]): String = seq match {
  case head1 +: head2 +: tail => s"($head1, $head2), " + windows2(seq.tail)
  case head +: tail => s"($head, _), " + windows2(tail)
  case Nil => "Nil"
}

for (seq <- Seq(nonEmptyList, emptyList, nonEmptyMap.toSeq)) {
  println(windows2(seq))
}
```

滑动窗口如此有用，Seq 甚至提供了两个方法用于创建窗口：

```
scala> val seq = Seq(1,2,3,4,5)
seq: Seq[Int] = List(1, 2, 3, 4, 5)

scala> val slide2 = seq.sliding(2)
slide2: Iterator[Seq[Int]] = non-empty iterator

scala> slide2.toSeq
res0: Seq[Seq[Int]] = res56: Seq[Seq[Int]] = Stream(List(1, 2), ?)

scala> slide2.toList
res1: List[Seq[Int]] = List(List(1, 2), List(2, 3), List(3, 4), List(4, 5))

scala> seq.sliding(3,2).toList
res2: List[Seq[Int]] = List(List(1, 2, 3), List(3, 4, 5))
```

这两个 sliding 方法都返回迭代器，它们是“惰性”的。由于对大的序列进行复制代价太过昂贵，这两个函数都没有立即对所操作的列表进行复制。对返回的迭代器调用 toSeq 方法，可以将迭代器转为一个 collection.immutable.Stream（http://www.scala-lang.org/api/current/#scala.collection.immutable.Stream）。这是一个惰性列表，创建时即对列表的头部元素求值，但只在需要的时候才对尾部元素求值。调用 toList 不一样，它返回一个 List，创建时就求出了所有元素的值。

注意，输出的结果与之前的例子有一点点不同，例如，在这里输出的结尾没有 (5, _)。

## 4.7 可变参数列表的匹配

在 2.6 节，我们介绍了 Scala 对方法中的可变参数列表的支持。例如，在编写一个与 SQL 交互的工具，或是用一个 case 类来表示 WHERE foo IN (val1, val2, …) 这样的 SQL 语句（这个例子来自于真实项目而非开源代码的启发）时，我们用到了 Scala 对可变参数列表的支持。以下就是一个带可变参数列表的 case 类，用于处理 SQL 语句中的各个值。代码中还包括了另外一些定义，用于处理 WHERE x OP y 这样的 SQL 语句，OP 是 SQL 的比较操作符。

```
// src/main/scala/progscala2/patternmatching/match-vararglist.sc
```

```
// Operators for WHERE clauses
object Op extends Enumeration {                                   // ❶
  type Op = Value

  val EQ   = Value("=")
  val NE   = Value("!=")
  val LTGT = Value("<>")
  val LT   = Value("<")
  val LE   = Value("<=")
  val GT   = Value(">")
  val GE   = Value(">=")
}
import Op._

// 表示 SQL的 "WHERE x op value"语句,其中+op+为一个比较
// 操作符: =, !=, <>, <, <=, >, or >=。
case class WhereOp[T](columnName: String, op: Op, value: T)        // ❷

// 表示SQL的"WHERE x IN (a, b, c, ...)" 语句。
case class WhereIn[T](columnName: String, val1: T, vals: T*)       // ❸

val wheres = Seq(                                                 // ❹
  WhereIn("state", "IL", "CA", "VA"),
  WhereOp("state", EQ, "IL"),
  WhereOp("name", EQ, "Buck Trends"),
  WhereOp("age", GT, 29))

for (where <- wheres) {
  where match {
    case WhereIn(col, val1, vals @ _*) =>                         // ❺
      val valStr = (val1 +: vals).mkString(", ")
      println (s"WHERE $col IN ($valStr)")
    case WhereOp(col, op, value) => println (s"WHERE $col $op $value")
    case _ => println (s"ERROR: Unknown expression: $where")
  }
}
```

❶ 定义了一个 Enumeration 用于表示比较 SQL 操作符，每个操作符有一个“名字”，是一个字符串。

❷ 用于表示 `WHERE x OP y` 的 case 类。

❸ 用于表示 `WHERE x IN (val1, val2, …)` 的 case 类。

❹ 用于解析的示例对象。

❺ 注意匹配可变参数的语法形式 : `name @ _*`。

用于匹配可变参数列表的语法 `name @ _*` 并不太符合直觉，但在有些场合你的确需要它。以下是示例运行的输出：

```
WHERE state IN (IL, CA, VA)
WHERE state = IL
WHERE name = Buck Trends
WHERE age > 29
```

# 4.8 正则表达式的匹配

正则表达式可以很方便地从符合特定结构的字符串中提取数据。

Scala 封装了 Java 的正则表达式[1]。以下给出一个示例：

```
// src/main/scala/progscala2/patternmatching/match-regex.sc

val BookExtractorRE = """Book: title=([^,]+),\s+author=(.+)""".r     // ❶
val MagazineExtractorRE = """Magazine: title=([^,]+),\s+issue=(.+)""".r

val catalog = Seq(
  "Book: title=Programming Scala Second Edition, author=Dean Wampler",
  "Magazine: title=The New Yorker, issue=January 2014",
  "Unknown: text=Who put this here??"
)

for (item <- catalog) {
  item match {
    case BookExtractorRE(title, author) =>                          // ❷
      println(s"""Book "$title", written by $author""")
    case MagazineExtractorRE(title, issue) =>
      println(s"""Magazine "title", issue $issue""")
    case entry => println(s"Unrecognized entry: $entry")
  }
}
```

❶ 该正则表达式匹配一个用于表示书本的字符串，其中有两个捕捉组（注意正则表达式中的括号），一个表示标题，一个表示作者。调用 `r` 方法以创建正则表达式。第二个正则表达式匹配一个用于表示杂志的字符串，其中的捕捉组表示杂志标题和发行时间。

❷ 用法与 `case` 类相似，与捕捉组相匹配的字符串被提取出来，赋值给变量。

运行的输出为：

```
Book "Programming Scala Second Edition", written by Dean Wampler
Magazine "The New Yorker", issue January 2014
Unrecognized entry: Unknown: text=Who put this here??
```

我们用三重双引号来表示正则表达式字符串，否则，就不得不对正则表达式的反斜杠进行转义，例如用 `\\s` 表示 `\s`。你还可以通过创建一个 `Regex` 类的实例来定义正则表达式，如 `new Regex("""\W""")`，但这种用法并不常见。

在三个双引号内的正则表达式中使用变量插值是无效的。你依然需要对变量插值进行转义，例如，你应该用 `s"""$first\\s+$second""".r`，而不是 `s"""$first\s+$second""".r`。而如果你没有使用变量插值，则不必转义。

`scala.util.matching.Regex` 定义了若干个用于正则表达式其他操作的方法，如查找和替换。

注 1：参见《Java 教程：正则表达式》。

## 4.9　再谈case语句的变量绑定

假设以下场景：你需要从对象中提取值，但你又想将一个变量赋给该对象的整体。该怎么做呢？

我们来对前文中匹配 Person 类的属性的实例做以下修改。

```
// src/main/scala/progscala2/patternmatching/match-deep2.sc

case class Address(street: String, city: String, country: String)
case class Person(name: String, age: Int, address: Address)

val alice   = Person("Alice",   25, Address("1 Scala Lane", "Chicago", "USA"))
val bob     = Person("Bob",     29, Address("2 Java Ave.",  "Miami",   "USA"))
val charlie = Person("Charlie", 32, Address("3 Python Ct.", "Boston",  "USA"))

for (person <- Seq(alice, bob, charlie)) {
  person match {
    case p @ Person("Alice", 25, address) => println(s"Hi Alice! $p")
    case p @ Person("Bob", 29, a @ Address(street, city, country)) =>
      println(s"Hi ${p.name}! age ${p.age}, in ${a.city}")
    case p @ Person(name, age, _) =>
      println(s"Who are you, $age year-old person named $name? $p")
  }
}
```

p @ …的语法将整个 Person 类的实例赋值给了变量 p，类似地，a @ …也将整个 Address 实例赋值给了变量。以下为修改版本的实例输出。（为适应排版做了格式化。）

```
Hi Alice! Person(Alice,25,Address(1 Scala Lane,Chicago,USA))
Hi Bob! age 29, in Miami
Who are you, 32 year-old person named Charlie? Person(Charlie,32,
  Address(3 Python Ct.,Boston,USA))
```

记住，如果不需要从 Person 实例中提取属性值，我们只要写为 p: Person => …就可以了。

## 4.10　再谈类型匹配

考虑以下例子，我们试图将输入的 List[Double] 和 List[String] 区分开：

```
// src/main/scala/progscala2/patternmatching/match-types.sc
scala> for {
     |   x <- Seq(List(5.5,5.6,5.7), List("a", "b"))
     | } yield (x match {
     |   case seqd: Seq[Double] => ("seq double", seqd)
     |   case seqs: Seq[String] => ("seq string", seqs)
     |   case _                 => ("unknown!", x)
     | })
<console>:12: warning: non-variable type argument Double in type pattern
Seq[Double] (the underlying of Seq[Double]) is unchecked since it is
eliminated by erasure
                case seqd: Seq[Double] => ("seq double", seqd)
                           ^
<console>:13: warning: non-variable type argument String in type pattern
```

```
Seq[String] (the underlying of Seq[String]) is unchecked since it is
eliminated by erasure
                case seqs: Seq[String] => ("seq string", seqs)
                         ^
<console>:13: warning: unreachable code
                case seqs: Seq[String] => ("seq string", seqs)
                                        ^
res0: List[(String, List[Any])] =
  List((seq double,List(5.5, 5.6, 5.7)),(seq double,List(a, b)))
```

这些警告表示什么？ Scala 运行于 JVM 中，这些警告来源于 JVM 的类型擦除，类型擦除是 Java 5 引入泛型后的一个历史遗留。为了避免与旧版本代码断代，JVM 的字节码不会记住一个泛型实例（如 List）中实际传入的类型参数的信息。

所以，当编译器只能识别输入对象为 List，但无法在运行时识别它是 List[Double] 还是 List[String] 时，编译器就会发出警告。事实上，编译器认为第二个匹配 List[String] 的 case 子句是不可达代码，意味着第一个匹配 List[Double] 的 case 子句可以匹配任意 List。输出显示，对于两个输入，都打印出了 seq double。

一个不太美观但却有效的解决方法是：首先匹配集合，然后用一个嵌套的匹配语句去匹配集合中的第一个元素，从而决定其类型。这样的话，我们也就必须单独处理空序列：

```
// src/main/scala/progscala2/patternmatching/match-types2.sc

def doSeqMatch[T](seq: Seq[T]): String = seq match {
  case Nil => "Nothing"
  case head +: _ => head match {
    case _ : Double => "Double"
    case _ : String => "String"
    case _ => "Unmatched seq element"
  }
}

for {
  x <- Seq(List(5.5,5.6,5.7), List("a", "b"), Nil)
} yield {
  x match {
    case seq: Seq[_] => (s"seq ${doSeqMatch(seq)}", seq)
    case _           => ("unknown!", x)
  }
}
```

以上脚本返回了期望的输出:Seq((seq Double,List(5.5, 5.6, 5.7)), (seq String,List(a, b)), (seq Nothing,List()))。

# 4.11 封闭继承层级与全覆盖匹配

这一小节我们来讨论全覆盖匹配的需求和封闭类层级的应用场景。其实，我们在 2.10 节已经对封闭类做了介绍。我们用在以下代码中使用封闭继承层级类来表示 HTTP 协议的合法消息类型（或称为“方法”）。

```
// src/main/scala/progscala2/patternmatching/http.sc
```

```
sealed abstract class HttpMethod() {                          // ❶
    def body: String                                          // ❷
    def bodyLength = body.length
}

case class Connect(body: String) extends HttpMethod           // ❸
case class Delete (body: String) extends HttpMethod
case class Get    (body: String) extends HttpMethod
case class Head   (body: String) extends HttpMethod
case class Options(body: String) extends HttpMethod
case class Post   (body: String) extends HttpMethod
case class Put    (body: String) extends HttpMethod
case class Trace  (body: String) extends HttpMethod

def handle (method: HttpMethod) = method match {              // ❹
  case Connect (body) => s"connect: (length: ${method.bodyLength}) $body"
  case Delete  (body) => s"delete:  (length: ${method.bodyLength}) $body"
  case Get     (body) => s"get:     (length: ${method.bodyLength}) $body"
  case Head    (body) => s"head:    (length: ${method.bodyLength}) $body"
  case Options (body) => s"options: (length: ${method.bodyLength}) $body"
  case Post    (body) => s"post:    (length: ${method.bodyLength}) $body"
  case Put     (body) => s"put:     (length: ${method.bodyLength}) $body"
  case Trace   (body) => s"trace:   (length: ${method.bodyLength}) $body"
}

val methods = Seq(
  Connect("connect body..."),
  Delete ("delete body..."),
  Get    ("get body..."),
  Head   ("head body..."),
  Options("options body..."),
  Post   ("post body..."),
  Put    ("put body..."),
  Trace  ("trace body..."))

methods foreach (method => println(handle(method)))
```

❶ 定义了一个封闭的抽象基类。由于该类被定义为封闭的，其子类型必须定义在本文件内。

❷ 为 HTTP 报文的消息体定义了一个方法。

❸ 定义了 8 个继承自 `HttpMethod` 的 case 类，每个类均在构造方法中声明了参数 `body: String`。由于每个类均为 case 类，因此该参数是一个 `val`，它实现了 `HttpMethod` 的抽象方法 `def`。

❹ 这是一个全覆盖的模式匹配表达式。即使我们没有默认的匹配子句，也可以达到全覆盖，因为输入的参数 `method` 只可能是我们定义的 8 个 case 类的实例。

> 对封闭基类的实例做模式匹配时，如果 `case` 语句覆盖了所有当前文件定义的类型，那么匹配就是全覆盖的。由于不允许有其他用于自定义的子类型，随着项目演进，匹配的全覆盖性也不会丧失。

由此可以得出的一个推论：如果类型的继承层级可能发生变化，就应该避免使用 sealed。这取决于你原有的面向对象继承规则，包括多态方法的设计情况。如果你去掉 HttpMethod 的 sealed 关键字，然后在本文件或其他文件新定义一个子类型，会怎么样呢？你必须在代码库以及客户端的代码库中找出并修改所有关于 HttpMethod 实例的模式匹配代码。

另外，这里还给出了一种实现某些方法的技巧。一个抽象的，不带参数的父类方法，在子类型中可以用一个 val 实现。这是由于 val 的值是固定的（必定的），而一个不带参数、返回值为某类型变量的方法可以返回任意一个该类型的变量。这样，使用 val 实现的方法在返回值上严格符合方法定义，当方法被“调用”时，使用 val 变量与真实调用方法一样安全。事实上，这是透明引用的一个应用。在透明引用中，我们用一个值代替一个总是返回固定值的表达式！

在父类型中，不带参数的抽象方法可以在子类中用 val 变量实现。推荐的做法是：在抽象父类型中声明一个不带参数的抽象方法，这样就给子类型如何具体实现该方法留下了巨大的自由，既可以用方法实现，也可以用 val 变量实现。

执行该脚本，产生以下输出：

```
connect: (length: 15) connect body...
delete:  (length: 14) delete body...
get:     (length: 11) get body...
head:    (length: 12) head body...
options: (length: 15) options body...
post:    (length: 12) post body...
put:     (length: 11) put body...
trace:   (length: 13) trace body...
```

HttpMethod 的 case 类很小，理论上我们也可以用 Enumeration 代替。但那样会有一个很大的缺陷，就是编译器就无法判断 Enumeration 相应的 match 语句是否全覆盖。如果我们在这里使用了 Enumeration，而在 match 语句中忘记了匹配 Trace 的语句，那我们也只能在运行时抛出 MatchError 的时候才知道这个错误的存在。

当使用类型匹配时避免使用枚举类型。编译器无法判断匹配语句是否全覆盖。

## 4.12 模式匹配的其他用法

幸运的是，模式匹配这一强大特性并不仅仅局限于 case 语句。在定义变量时也可以运用模式匹配，包括 for 表达式中的变量定义。

```
scala> case class Address(street: String, city: String, country: String)
defined class Address

scala> case class Person(name: String, age: Int, address: Address)
defined class Person
```

```
scala> val Person(name, age, Address(_, state, _)) =
     |   Person("Dean", 29, Address("1 Scala Way", "CA", "USA"))
name: String = Dean
age: Int = 29
state: String = CA
```

没错，只用了一个步骤，我们就将 Person 中所有需要的属性抽取了出来，同时略过了不需要的属性。这个方法也可以用在 List 上。

```
scala> val head +: tail = List(1,2,3)
head: Int = 1
tail: List[Int] = List(2, 3)

scala> val head1 +: head2 +: tail = Vector(1,2,3)
head1: Int = 1
head2: Int = 2
tail: scala.collection.immutable.Vector[Int] = Vector(3)

scala> val Seq(a,b,c) = List(1,2,3)
a: Int = 1
b: Int = 2
c: Int = 3

scala> val Seq(a,b,c) = List(1,2,3,4)
scala.MatchError: List(1, 2, 3, 4) (of class collection.immutable.$colon$colon)
  ... 43 elided
```

这个技巧非常好用。在你自己的示例中尝试运用下吧。

在 if 表达式中我们也可以用模式匹配：

```
scala> val p = Person("Dean", 29, Address("1 Scala Way", "CA", "USA"))
p: Person = Person(Dean,29,Address(1 Scala Way,CA,USA))

scala> if (p == Person("Dean", 29,
     |    Address("1 Scala Way", "CA", "USA"))) "yes" else "no"
res0: String = yes

scala> if (p == Person("Dean", 29,
     |    Address("1 Scala Way", "CA", "USSR"))) "yes" else "no"
res1: String = no
```

然而，在这里无法使用 _ 占位符：

```
scala> if (p == Person(_, 29, Address(_, _, "USA"))) "yes" else "no"
<console>:13: error: missing parameter type for expanded function
 ((x$1) => p.$eq$eq(Person(x$1,29,((x$2,x$3) => Address(x$2,x$3,"USA")))))
             if (p == Person(_, 29, Address(_, _, "USA"))) "yes" else "no"
                                ^
 ...
```

名为 $eq$eq 的内部函数用于 == 测试。因为在 JVM 规范中，只允许字母、数字、_ 和 $ 作为标识符，Scala 对一些非字母数字的字符做了“字符映射”，使得它们符合 JVM 规范。在这里，= 变成了 $eq。所有的映射规则会在第 22 章的表 22-1 中列出。

假设我们有一个函数，参数为整数序列，将所有整数的和与整数的个数放在元组中返回：

```
scala> def sum_count(ints: Seq[Int]) = (ints.sum, ints.size)

scala> val (sum, count) = sum_count(List(1,2,3,4,5))
sum: Int = 15
count: Int = 5
```

这个用法我经常使用。在 3.6.5 节有一个人为生造的例子，它在 for 表达式中使用模式匹配。以下给出了该例子中与模式匹配相关的片段：

```
// src/main/scala/progscala2/patternmatching/scoped-option-for.sc

val dogBreeds = Seq(Some("Doberman"), None, Some("Yorkshire Terrier"),
                    Some("Dachshund"), None, Some("Scottish Terrier"),
                    None, Some("Great Dane"), Some("Portuguese Water Dog"))

println("second pass:")
for {
  Some(breed) <- dogBreeds
  upcasedBreed = breed.toUpperCase()
} println(upcasedBreed)
```

如同前文，脚本输出依然为：

```
DOBERMAN
YORKSHIRE TERRIER
DACHSHUND
SCOTTISH TERRIER
GREAT DANE
PORTUGUESE WATER DOG
```

模式匹配与 case 语句的另一个便利用法是，它们可以使带复杂参数的函数字面量更易于使用：

```
// src/main/scala/progscala2/patternmatching/match-fun-args.sc

case class Address(street: String, city: String, country: String)
case class Person(name: String, age: Int)

val as = Seq(
  Address("1 Scala Lane", "Anytown", "USA"),
  Address("2 Clojure Lane", "Othertown", "USA"))
val ps = Seq(
  Person("Buck Trends", 29),
  Person("Clo Jure", 28))

val pas = ps zip as

// 不太美观的方法：
pas map { tup =>
  val Person(name, age) = tup._1
  val Address(street, city, country) = tup._2
  s"$name (age: $age) lives at $street, $city, in $country"
}
```

```
// 不错的方法：
pas map {
  case (Person(name, age), Address(street, city, country)) =>
    s"$name (age: $age) lives at $street, $city, in $country"
}
```

注意，压缩序列的类型为 Seq[(Person,Address)]，所以，我们传递给 map 的参数必须是一个类型为 (Person,Address) => String 的函数。我们给出了两个函数，第一个是一个“常规”函数，带一个元组参数，使用模式匹配从元组的两个元素中提取属性值。

第二个函数是一个偏函数，这在偏函数中有介绍。这种写法在语法上更为简洁，特别是需要从元组和更复杂的结构中抽取值时。但是需要记住：由于给出的是一个偏函数，所以 case 表达式必须精确匹配输入，否则在运行时会抛出一个 MatchError。

使用以上两个函数，产生的字符串序列均为：

```
List(
  "Buck Trends (age: 29) lives at 1 Scala Lane, Anytown, in USA",
  "Clo Jure (age: 28) lives at 2 Clojure Lane, Othertown, in USA")
```

最后要介绍的是，我们可以在正则表达式中用模式匹配去解构字符串。我写过一份用于 SQL 解析的简单程序，以下是从该程序的测试代码中抽取的例子：

```
// src/main/scala/progscala2/patternmatching/regex-assignments.sc
scala> val cols = """\*|[\w, ]+""" // 用于提取列
cols: String = \*|[\w, ]+

scala> val table = """\w+"""       // 用于提取表
table: String = \w+

scala> val tail = """.*"""         // 用于其他语句
tail: String = .*

scala> val selectRE =
     | s"""SELECT\\s*(DISTINCT)?\\s+($cols)\\s*FROM\\s+($table)\\s*($tail)?;""".r
selectRE: scala.util.matching.Regex = \
  SELECT\s*(DISTINCT)?\s+(\*|[\w, ]+)\s*FROM\s+(\w+)\s*(.*)?;

scala> val selectRE(distinct1, cols1, table1, otherClauses) =
     |   "SELECT DISTINCT * FROM atable;"
distinct1: String = DISTINCT
cols1: String = *
table1: String = atable
otherClauses: String = ""

scala> val selectRE(distinct2, cols2, table2, otherClauses) =
     |   "SELECT col1, col2 FROM atable;"
distinct2: String = null
cols2: String = "col1, col2 "
table2: String = atable
otherClauses: String = ""

scala> val selectRE(distinct3, cols3, table3, otherClauses) =
```

```
     |   "SELECT DISTINCT col1, col2 FROM atable;"
distinct3: String = DISTINCT
cols3: String = "col1, col2 "
table3: String = atable
otherClauses: String = ""

scala> val selectRE(distinct4, cols4, table4, otherClauses) =
     |   "SELECT DISTINCT col1, col2 FROM atable WHERE col1 = 'foo';"
distinct4: String = DISTINCT
cols4: String = "col1, col2 "
table4: String = atable
otherClauses: String = WHERE col1 = 'foo'
```

注意，由于用了变量插值，因此在正则表达式字符串中，必须增加反斜杠进行转义，如用 `\\s` 代替 `\s`。

显然，用正则表达式去解析复杂文本如 XML 或编程语言，有其局限性。抛开简单的例子，我们考虑一个解析库就会明白正则表达式的局限性。我们会在第 20 章进一步讨论。

## 4.13 总结关于模式匹配的评价

模式匹配是一个强大的“协议”，用于从数据结构中提取数据。JavaBeans 模型有一个无意中造成的结果，就是模式匹配将会鼓励开发者用 getter 和 setter 暴露对象的属性。而这种做法往往忽略了一点，即状态应该被封装，只在恰当的时候才暴露出来，尤其对可变的属性而言更是如此。对状态信息的获取应该小心设计，以反映暴露的抽象。

考虑一下，当你需要以一种可控的方式抽取信息时，如何使用模式匹配？正如我们在 `unapply` 方法中看到的那样，你可以自定义 `unapply` 方法去控制暴露出来的状态。这些方法允许你抽取信息的同时隐藏实现细节。实际上，`unapply` 方法返回的信息可能是类型的属性值经过转换后的结果。

最后要说明的是：当设计模式匹配语句时，需要谨慎对待默认的 `case` 子句。在什么情况下，才应该出现“以上均不匹配”的情况呢？默认 `case` 子句有可能表明，你该改善一下程序的设计了。这样，你会更准确地知道程序中可能发生的所有匹配的情况。

在 `for` 循环中，模式匹配的惯用方法使得 Scala 代码简单却又强大。对于功能相同的代码，Scala 程序的行数比 Java 程序少 10 倍的情况并不罕见。

所以，尽管 Java 8 增加匿名函数（Lambda）——一种类似模式匹配和 `for` 循环的工具——是一个巨大的改进，但 Scala 可以缩减几行代码的优势，是一个让你由 Java 转向 Scala 的理由。

## 4.14 本章回顾与下一章提要

模式匹配是很多函数式语言之所以强大的标志。它是一个可以灵活又简洁地从数据结构中抽取数据的工具。我们已经看到了在 `case` 语句和其他表达式中使用模式匹配的示例。

下一章我们将讨论一个 Scala 中独一无二、强大但有争议的特性——隐含。其中有一系列用于构建领域特定语言（DSL）的工具，可以减少冗余度，使 API 更易使用、更易进行自定义。

# 第 5 章

# 隐式详解

隐式（implicit）是 Scala 的一个强大的特性，同时也是一个可能存在争议的特性。使用隐式能够减少代码，能够向已有类型中注入新的方法，也能够创建领域特定语言（DSL）。

隐式之所以会产生争议，是因为除了通过 `Predef` 对象自动加载的那些隐式对象外，其他在源码中出现的隐式对象均不是本地对象。隐式对象一旦进入作用域，编译器便能执行该隐式对象以生成方法参数或将指定参数转化成预期类型。不过在阅读源代码时，读者无法简单地指出什么时候会应用这些隐式值和隐式方法，而这可能会给该读者造成困惑。幸运的是随着经验的累积，你能够意识到什么时候将会触发这些隐式对象，你也可以通过阅读这些对象的 API 来学习这些知识。不过对于初学者而言，这将是一段意外之旅。

想要理解隐式对象的工作机制需要采用较为直接的学习方法。本章的大多数篇幅将通过示例讲解隐式对象能够解决的问题。

## 5.1 隐式参数

在 2.5.3 节中，我们使用了 `implicit` 关键字标记那些用户无需显式提供的方法参数。调用方法时，如果未输入隐式参数且代码所处作用域中存在类型兼容值时，类型兼容值会从作用域中调出并被使用，反之，系统将会抛出编译器错误。

假设我们定义了一个用于计算销售税的方法，而税率被设置为隐式参数。

```
def calcTax(amount: Float)(implicit rate: Float): Float = amount * rate
```

调用该方法时，系统会将代码所在局部作用域中的某一隐式值传入此方法：

```
implicit val currentTaxRate = 0.08F
...
val tax = calcTax(50000F)  // 4000.0
```

对于某些简单的场景，设定一个固定的浮点数就能满足需求。不过有些场景可能就不这么简单了。例如某些应用需要知道当前事务发生的具体地点，以便增收地方税。而为了促进购物消费，某些辖区也可能会将年假的最后几天设定为“免税期”。

幸运的是，我们可以使用隐式方法解决这一问题。隐式对象本身不具有任何参数，除非该参数同样被标示为隐式参数。下面列举了计算销售税的完整示例：

```
// src/main/scala/progscala2/implicits/implicit-args.sc

// 永远不要用FLoat类型表示货币:
def calcTax(amount: Float)(implicit rate: Float): Float = amount * rate

object SimpleStateSalesTax {
  implicit val rate: Float = 0.05F
}

case class ComplicatedSalesTaxData(
  baseRate: Float,
  isTaxHoliday: Boolean,
  storeId: Int)

object ComplicatedSalesTax {
  private def extraTaxRateForStore(id: Int): Float = {
    // 可以通过id推断出商铺所在地,之后再计算附加税……
    0.0F
  }

  implicit def rate(implicit cstd: ComplicatedSalesTaxData): Float =
    if (cstd.isTaxHoliday) 0.0F
    else cstd.baseRate + extraTaxRateForStore(cstd.storeId)
}

{
  import SimpleStateSalesTax.rate

  val amount = 100F
  println(s"Tax on $amount = ${calcTax(amount)}")
}

{
  import ComplicatedSalesTax.rate
  implicit val myStore = ComplicatedSalesTaxData(0.06F, false, 1010)

  val amount = 100F
  println(s"Tax on $amount = ${calcTax(amount)}")
}
```

尽管我们是在可插入字符串（interpolated string）中调用 calcTax 方法，但该方法仍然会将隐式值应用到 rate 参数上。

还有一类更复杂的情景：我们可以定义一个包含了隐式参数的隐式方法，该隐式参数将接收方法所需的数据。

运行该脚本将得到以下输出：

```
Tax on 100.0 = 5.0
Tax on 100.0 = 6.0
```

### 调用 implicitly方法

Predef 对象中定义了一个名为 implicity 的方法。如果将 implicity 方法与附加类型签名（type signature addition）相结合，便能以一种有用且快捷的方式定义一个接收参数化类型隐式参数的函数。

在下列示例中，我们使用了这种方法对 List 的 sortBy 方法进行封装。

```
// src/main/scala/progscala2/implicits/implicitly-args.sc
import math.Ordering

case class MyList[A](list: List[A]) {
  def sortBy1[B](f: A => B)(implicit ord: Ordering[B]): List[A] =
    list.sortBy(f)(ord)

  def sortBy2[B : Ordering](f: A => B): List[A] =
    list.sortBy(f)(implicitly[Ordering[B]])
}

val list = MyList(List(1,3,5,2,4))

list sortBy1 (i => -i)
list sortBy2 (i => -i)
```

有些集合提供了一些排序方法，List.sortBy 便是其中之一。List.sortBy 方法的第一个参数类型为函数，该输入函数能够将函数的输入参数转化为另一个满足 math.Ordering 条件的类型。而 math.Ordering 是与 Java 中的 Comparable（http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html）抽象等同的类型。List.sortBy 方法的另一个参数则为隐式参数，该参数知道如何对类型 B 的实例进行排序。

MyList 类提供了两种方式编写像 sortBy 类型的方法。第一种实现：sortBy1 方法应用了我们已知的语法。该方法接受一个额外的类型为 Ordering[B] 的隐式值作为其输入。调用 sortBy1 方法时，在当前作用域中一定存在某一 Ordering[B] 的对象实例，该实例清楚地知道如何对我们所需要的 B 类型对象进行排序。我们认为 B 的界限被“上下文”所限定，在这个例子中，上下文限定了 B 对实例进行排序的能力。

由于这种 Scala 方言应用非常普遍，因此 Scala 提供了一个简化版的语法，这正是第二类实现 sortBy2 所使用的语法。类型参数 B : Ordering 被称为*上下文定界*（context bound），它暗指第二个参数列表（也就是那个隐式参数列表）将接受 Ordering[B] 实例。

不过，我们仍需要在方法中访问 Ordering 对象实例。由于在源代码中我们不再明确地声明 Ordering 对象实例，因此这个实例没有自己的名称。针对这种现象我们该怎么办呢？Predef.implicitly 方法帮我们解决了这个问题。implicitly 方法会对传给函数的所有标记为隐式参数的实例进行解析。请注意 implicitly 方法所需要的类型签名，在本例中 Ordering[B] 是其类型签名。

当我们需要类型为参数化类型的隐式参数时，当类型参数属于当前作用域的其他一些类型时（例如：[B : Ordering] 代表了类型为 Ordering[B] 的隐式参数），可以将上下文定界（context bound）与 implicitly 方法结合起来，以简洁地方式解决这个问题。

## 5.2　隐式参数适用的场景

程序员应谨慎明智地使用隐式对象。滥用隐式对象会让读者无法理解代码的用意。

既然隐式参数具有弊端，那么我们为什么还要使用它呢？目前已经有不少通过隐式参数实现的*常见习语*（idiom），隐式参数为这些习语带来两类好处：第一类好处是能够消除样板代码，例如隐式对象提供了上下文信息以省略明确指定这些信息的代码；第二类好处是通过引入约束来减少 bug 数量以及使用参数化类型对某些方法允许的输入参数类型进行限定。下面我们将对这些习语进行分析。

### 5.2.1　执行上下文

在 2.5.3 节，我们学习了关于 Future 对象的示例，示例中的 apply 方法的第二个参数列表被设为隐式参数，该参数用于将 ExecutionContext 对象（http://www.scala-lang.org/api/current/#scala.concurrent.ExecutionContext）传递给 Future.apply 方法（http://www.scala-lang.org/api/current/#scala.concurrent.Future$）：

```
apply[T](body: => T)(implicit executor: ExecutionContext): Future[T]
```

一些其他的方法同样提供了这类隐式参数。

尽管我们在调用这些方法时并未指定 ExecutionContext 对象，但是我们导入了可供编译器使用的全局默认值：

```
import scala.concurrent.ExecutionContext.Implicits.global
```

使用隐式参数传入“执行上下文”是一个值得推荐的用法。另外，编写事务、数据库连接、线程池以及用户会话时隐式参数上下文也同样适合使用。使用方法参数能组合行为，而将方法参数设置为隐式参数能够使 API 变得更加简洁。

### 5.2.2　功能控制

除了能够传递上下文对象之外，隐式参数还具有控制系统功能的作用。

举个例子，通过引入授权令牌，我们可以控制某些特定的 API 操作只能供某些用户调用，我们也可以使用授权令牌决定数据可见性，而隐式用户会话参数也许就包含了这类令牌信息。

假设你想要创建用户界面的菜单，其中某些菜单项只对已登录用户可见，而其他菜单项则仅对未登录用户可见。

```
def createMenu(implicit session: Session): Menu = {
  val defaultItems = List(helpItem, searchItem)
```

```
    val accountItems =
      if (session.loggedin()) List(viewAccountItem, editAccountItem)
      else List(loginItem)
    Menu(defaultItems ++ accountItems)
  }
```

### 5.2.3 限定可用实例

设想一下，我们希望对具有参数化类型方法中的类型参数进行限定，使该参数只接受某些类型的输入。那么该如何处理呢？

假如允许输入的所有参数类型均为某一公共超类的子类型，那么无需应用隐式技术，面向对象的技术便可解决这一问题。让我们首先思考一下解决方案。

在 3.10 节中，我们通过示例演示掌握了如何实现资源管理器：

```
object manage {
  def apply[R <: { def close():Unit }, T](resource: => R)(f: R => T) = {...}
  ...
 }
```

类型参数 `R` 必须是定义了 `close():Unit` 方法的任意类型的子类。否则，我们管理的所有资源都必须实现 `Closable` 的 trait（请回顾 Scala 是如何使用 trait 取代并扩展 Java 接口的。详情参见 3.14 节）：

```
trait Closable {
  def close(): Unit
}
...
object manage {
  def apply[R <: Closable, T](resource: => R)(f: R => T) = {...}
  ...
}
```

假如这些类型并无公共超类，这项技术便无用武之地。对于这种情况，我们可以使用隐式参数对允许的类型进行限定。Scala 集合 API 就是利用这项技术解决了一些设计上的问题。

具体的集合类所支持的某些方法是由父类实现的。例如：`List[A].map(f: A => B): List[B]` 方法首先将函数 `f` 运用在各个元素之上，之后又创建了一个新的列表。 由于大多数的集合都提供了 `map` 方法，因此在一个通用的 trait 中实现 `map` 方法，再将该 trait 混入（我们在 3.14 节中讨论了如何使用 trait 实现“混入”）到需要该方法的集合中也就顺理成章了。不过假如我们希望 `map` 方法能返回与容器元素类型相同的类型，那么又该使用什么机制通知 `map` 方法呢？

#### 应用Scala API

Scala API 运用一种常见的手法，将一个“构建器”（builder）作为隐式参数传入到 `map` 方法中。该构建器知道如何构造一个同种类型的新容器。这实际上与定义在 `TraversableLike`（http://www.scala-lang.org/api/current/index.html#scala.collection.TraversableLike） 中的 `map` 方法的签名很相似。`TraversableLike` 是一个 trait，它被混入了那些“可遍历”

（traversable）的容器类型中。

```
trait TraversableLike[+A, +Repr] extends ... {
  ...
  def map[B, That](f: A => B)(
    implicit bf: CanBuildFrom[Repr, B, That]): That = {...}
  ...
}
```

我们再回顾一下，+A 意味着 TraversableLike[A] 类型是 A 类型的协变（covariant）；假如 B 类型是 A 类型的子类型，那么 TraversableLike[B] 类型也是 TraversableLike[A] 类型的子类型。

CanBuildFrom（http://www.scala-lang.org/api/current/index.html#scala.collection.generic.CanBuildFrom）便是我们所使用的构造器。只要存在一个隐式构建器对象，你便能够构建出任意一个你想要的新容器。为了强调这点，该构建器被命名为 CanBuildForm。

实际上，容器通过 Repr 类型持有内部的元素，而 B 类型则是函数 f 所生成的元素的类型。

B 便是我们想要创建的目标集合的类型参数。通常情况下，我们希望构建一个与输入类型相同新的容器，允许类型参数有所差异。也就是说，B 也许与 A 类型相同，也许不同。Scala API 为所有内置的容器类型提供了隐式的 CanBuildFrom 构造器。

因此，map 操作可以输出的集合类型是由当前存在的对应的 CanBuildFrom 构造器实例所决定的，而这些构造器在当前作用域被声明为隐式对象。假如你自定义了某些容器，而你希望能够复用像 TraversableLike.map 这样的方法实现，你需要创建 CanBuildFrom 类型，并在这些容器的代码中导入它们的隐式示例。

下面我们将学习另一个示例：你希望编写 Java 数据库 API 的 Scala 封装类。这个示例受到了 Cassandra 数据库 API（https://github.com/datastax/java-driver）的启发：

```
// src/main/scala/progscala2/implicits/java-database-api.scala

// 为了方便用户使用,我们使用Scala编写了数据库API,该API与Java API较为类似。
package progscala2.implicits {
  package database_api {

    case class InvalidColumnName(name: String)
      extends RuntimeException(s"Invalid column name $name")

    trait Row {
      def getInt   (colName: String): Int
      def getDouble(colName: String): Double
      def getText  (colName: String): String
    }
  }

  package javadb {
    import database_api._

    case class JRow(representation: Map[String,Any]) extends Row {
      private def get(colName: String): Any =
        representation.getOrElse(colName, throw InvalidColumnName(colName))
```

```
    def getInt   (colName: String): Int    = get(colName).asInstanceOf[Int]
    def getDouble(colName: String): Double = get(colName).asInstanceOf[Double]
    def getText  (colName: String): String = get(colName).asInstanceOf[String]
  }

  object JRow {
    def apply(pairs: (String,Any)*) = new JRow(Map(pairs :_*))
  }
 }
}
```

为了方便起见，我使用 Scala 语言编写了这个 API。API 使用 `Map` 表示结果集中的一行，不过考虑到效率问题，在现实的实现中也许会使用字节数组表示一行数据。

`getInt`、`getDouble`、`getText` 以及其他一些尚未实现的一组方法便构成了该 API 的核心功能。这些方法将某一列的原始数据转化为具有正确类型的值，假如你对某一列使用了错误的类型方法，这些方法将抛出 `ClassCaseException` 异常（http://docs.oracle.com/javase/8/docs/api/java/lang/ClassCastException.html）。

如果我们只定义一个 `get[T]` 方法，其中 `T` 代表某一允许的列值类型，会不会更好一些呢？这有助于提供更加统一的调用接口，因为调用这一方法时我们不再需要 case 语句选择正确的调用方法，而且有时候我们还能在这些接口中使用类型推导。

在 Java 中，原始类型与引用类型的区别之一在于我们无法在像 `get[T]` 这样的参数化方法中使用原始类型。我们必须使用装箱后的类型，比如使用 `java.lang.Integer` 类型来替代 `int` 类型。但是在高性能数据应用程序中我们往往不希望出现装箱操作的性能损耗！

不过，我们在 Scala 中可以这样做：

```
// src/main/scala/progscala2/implicits/scala-database-api.scala

// 运用Scala封装对象实现类Java的数据库API。
package progscala2.implicits {
    package scaladb {
    object implicits {
      import javadb.JRow

      implicit class SRow(jrow: JRow) {
        def get[T](colName: String)(implicit toT: (JRow,String) => T): T =
          toT(jrow, colName)
      }

      implicit val jrowToInt: (JRow,String) => Int =
        (jrow: JRow, colName: String) => jrow.getInt(colName)
      implicit val jrowToDouble: (JRow,String) => Double =
        (jrow: JRow, colName: String) => jrow.getDouble(colName)
      implicit val jrowToString: (JRow,String) => String =
        (jrow: JRow, colName: String) => jrow.getText(colName)
    }

    object DB {
      import implicits._
```

```
        def main(args: Array[String]) = {
          val row = javadb.JRow("one" -> 1, "two" -> 2.2, "three" -> "THREE!")

          val oneValue1: Int      = row.get("one")
          val twoValue1: Double   = row.get("two")
          val threeValue1: String = row.get("three")
          // val fourValue1: Byte    = row.get("four")  // 不编译该行

          println(s"one1   -> $oneValue1")
          println(s"two1   -> $twoValue1")
          println(s"three1 -> $threeValue1")

          val oneValue2   = row.get[Int]("one")
          val twoValue2   = row.get[Double]("two")
          val threeValue2 = row.get[String]("three")
          // val fourValue2    = row.get[Byte]("four")  // 不编译该行

          println(s"one2   -> $oneValue2")
          println(s"two2   -> $twoValue2")
          println(s"three2 -> $threeValue2")
        }
      }
    }
  }
```

在隐式对象中，我们使用了一个隐式类对 Java 的 JRow 类进行封装，而该封装类中提供了我们想要的 get[T] 方法。我们将这些类称为*隐式转换*（implicit conversion）。这一知识点我们将在本章后面的篇幅中谈论到。现在，你只需要了解使用隐式转换后我们可以对 JRow 实例调用 get[T] 方法，就好像该方法就是为该实例定义的那样。

get[T] 方法接受两个参数列表，第一个参数列表是从行中读取数据所需要使用到的列名，第二个是一个隐式函数参数。该函数将抽取行中某一列的数据，并将该列数据转化成正确的类型。

假如你仔细阅读 get[T] 方法，你会注意到该方法引用了 jrow 实例，而 jrow 实例被传递给了 SRow 的构造方法。不过，我们并未使用 val 关键字声明，因此 jrow 并不是该类的成员。那么 get[T] 方法是如何引用这个值的呢？很简单，由于 jrow 位于类体作用域内，get[T] 可以直接使用它。

> 有时某些构造参数并未被声明为一个字段（使用 val 或 var），这是因为这些参数并未持有类型的状态信息，因此也无需暴露给客户。不过由于这些参数位于整个类型体的作用域内，类型的其他成员仍可以引用它们。

我们接下来定义了三个函数类型的隐式值，这些函数输入 JRow，返回具有正确类型的列值。为了能够调用 get[T]，这些函数使用了 implicitly 方法。

最后，我们定义用于测试的 DB 对象。该对象首先创建了一个 JRow 对象，之后对 JRow 对象调用 get[T] 方法，获得了三列的数值。DB 对象执行了两遍这样的操作。第一遍执行操

作时，系统能够通过变量类型推导出 T 类型，例如：根据 oneValue1 的类型推导出 T 类型。而第二次执行操作时，我们省略了变量类型注释，明确地指定 get[T] 中 T 对应的参数值。因此，我更青睐于第二种方式。

如果想要执行该代码，我们需要启动 sbt 并输入 **run-main progscala2.implicits.scaladb.DB** 命令。该命令将会按需编译代码：

```
> run-main progscala2.implicits.scaladb.DB
[info] Running scaladb.DB
one1   -> 1
two1   -> 2.2
three1 -> THREE!
one2   -> 1
two2   -> 2.2
three2 -> THREE!
[success] Total time: 0 s, ...
```

请注意源代码中注释了抽取字节值对应的代码行。假如移除了这些行中的 // 字符，编译将会出现错误。下面列出了在移除第一行注释代码后的错误信息（为了适应图书大小，我们在错误信息中添加了换行符）。

```
[error] .../implicits/scala-database-api.scala:31: ambiguous implicit values:
[error]  both value jrowToInt in object Implicits of type =>
           (javadb.JRow, String) => Int
[error]  and value jrowToDouble in object Implicits of type =>
           (javadb.JRow, String) => Double
[error]  match expected type (javadb.JRow, String) => T
[error]       val fourValue1: Byte    = row.get("four")  // 本行不会被编译
```

我们遇到了两种可能错误中的一种。在这个例子中，由于作用域中存在一些隐式转换，同时，Byte 是如同 Int 或 Double 类型的数字，编译器因此并没有将其转换成两者中的任意一种类型，因为编译不允许出现二义性。但由于这两个函数抽取了太多的字节，无论如何都会抛出错误！

假设当前作用域中不存在任何隐式转换，你也会得到一个不同的错误。即使我们注释了对象 Implicit 中定义的三个隐式值，每次调用时也会出现下列错误：

```
[error] .../implicits/scala-database-api.scala:28:
  could not find implicit value for parameter toT: (javadb.JRow, String) => T
[error]       val oneValue1: Int      = row.get("one")
```

我们再回顾一下，通过传入一个隐式参数以及只定义符合我们允许的类型所对应的隐式值，我们就可以对可用于参数化方法的类型进行了限定。

顺便提一下，本示例受到了之前编写的 API 的启发。之前的 API 使得 JRow 能在更多时候与隐式函数关联上，这样我便可以嵌入“仿造”的或真实的 Cassandra 数据，其中的仿造数据用于测试。

### 5.2.4　隐式证据

在上一节中，我们讨论了如何使用隐式对象对允许的类型进行限定，而这些允许的类型并

不具有共同的超类。除此之外，我们还使用隐式对象执行 API 的相关工作。

有时候，我们只需要限定允许的类型，并不需要提供额外的处理。换句话说，我们需要“证据”证明提出的类型满足我们的需求。现在我们将讨论另外一种被称为隐式证据的相关技术来对允许的类型进行限定，而且这些类型无需继承某一共有的超类。

适用于所有*可遍历*（traversable）容器的 `topMap` 方法便是应用该技术的良好示例。我们回顾一下 `Map` 的构造函数，该函数接受*键－值对*（key-value pair）作为其输入参数，例如两元元组。假如我们拥有一连串的 pair，那么如果能通过一个操作基于这些 pair 值创建 `Map` 对象岂不是更好？这便是 `toMap` 方法做的事情，不过我们还是有所顾虑，因为我们并不允许在调用 `toMap` 方法时输入一组非 pair 类型的序列。

`TraversableOnce`（http://www.scala-lang.org/api/current/index.html#scala.collection.TraversableOnce）是这样定义 `toMap` 方法的：

```
trait TraversableOnce[+A] ... {
  ...
  def toMap[T, U](implicit ev: <:<[A, (T, U)]): immutable.Map[T, U]
  ...
}
```

隐式参数 `ev` 便是我们的“证据”，它代表了我们必须实施的约束。`ev` 运用了 `Predef` 中定义的名为 `<:<` 的类型，该名字取自于 `<:` 方法。`<:` 方法同样也被用于限定类型参数，例如：`A <: B`。

我们曾提及过，可以使用中缀表示法表示由两个类型参数所组成的类型，因此下列两种表达式是等价的：

```
<:<[A, B]
A <:< B
```

在 `toMap` 中，`B` 实际上是一个 pair：

```
<:<[A, (T, U)]
A <:< (T, U)
```

现在，假如我们希望将一个可遍历计划转换成 `Map` 对象，编译器会结合我们所需要的隐式证据 `ev` 值来进行判定，只有满足 `A <: (T,U)` 时才会执行操作。也就是说，编译器会检查 `A` 是否是一个 pair，如果满足该条件，便会调用 `toMap` 方法并将可遍历的元素传递给 `Map` 构造器；假如 `A` 不是 pair 类型，编译器将会抛出错误：

```
scala> val l1 = List(1, 2, 3)
l1: List[Int] = List(1, 2, 3)

scala> l1.toMap
<console>:9: error: Cannot prove that Int <:< (T, U).
              l1.toMap
                 ^

scala> val l2 = List("one" -> 1, "two" -> 2, "three" -> 3)
l2: List[(String, Int)] = List((one,1), (two,2), (three,3))
```

```
scala> l2.toMap
res3: scala.collection.immutable.Map[String,Int] =
  Map(one -> 1, two -> 2, three -> 3)
```

因此，“证据”存在的意义仅仅是为了实施某一类型约束。我们无需定义一个隐式值来执行额外的自定义工作。

Predef 对象中还定义了一个名为 =:= 的“证据”类型，它可以证明两个类型之间的等价关系。但该类型并未得到广泛的应用。

### 5.2.5 绕开类型擦除带来的限制

有了隐式“证据”之后，我们在计算时就可以不再使用隐式对象。更准确地说，我们只需要使用隐式“证据”证明输入满足某些特定的类型约束。

类型擦除（type erasure）会带来一些限制。但在接下来的示例中我们通过使用隐式对象提供的“证据”巧妙地绕开了这些限制。

由于历史原因，JVM“忘记”了为参数化类型提供类型参数。例如，我们在下面定义了重载方法，虽然这些方法具有相同的名称，但它们的类型签名却互不相同：

```
object C {
  def m(seq: Seq[Int]): Unit = println(s"Seq[Int]: $seq")
  def m(seq: Seq[String]): Unit = println(s"Seq[String]: $seq")
}
```

我们再来看看在 REPL 会话中运行这段代码会出现什么情况：

```
scala> :paste
// 进入粘贴模式(输入ctrl-D结束该模式)

object M {
    def m(seq: Seq[Int]): Unit = println(s"Seq[Int]: $seq")
    def m(seq: Seq[String]): Unit = println(s"Seq[String]: $seq")
}
<ctrl-d>

// 退出粘贴模式,现在进入解释执行模式。

<console>:8: error: double definition:
method m:(seq: Seq[String])Unit and
method m:(seq: Seq[Int])Unit at line 7
have same type after erasure: (seq: Seq)Unit
       def m(seq: Seq[String]): Unit = println(s"Seq[String]: $seq")
           ^
```

<ctrl-d> 说明我输入了 Ctrl-D，当然控制台并不会输出这一字符。

顺便提一下，假如在 object M 的作用域之外输入这两个方法的定义体，并且未使用 :paste 模式，那么你将看不到任何报错信息。这是因为为了让客户使用方便，REPL 允许重定义类型、值和方法，而编译器在编译常规文件时则不允许这点。假如你忘记了 REPL 中有这样一个“便民”的设计，你也许会以为你成功定义了两个不同版本的 m 方法。而运

行 :paste 模式时，编译器会把 Ctrl-D 出现前的所有输入视为一个待编译的普通文件。

所以，正因为这些方法在字节码中是一样的，编译器不允许同时出现这些方法定义。

不过，我们可以通过添加隐式参数来消除这些方法的二义性：

```
// src/main/scala/progscala2/implicits/implicit-erasure.sc
object M {
  implicit object IntMarker                                       // ❶
  implicit object StringMarker

  def m(seq: Seq[Int])(implicit i: IntMarker.type): Unit =        // ❷
    println(s"Seq[Int]: $seq")

  def m(seq: Seq[String])(implicit s: StringMarker.type): Unit =  // ❸
    println(s"Seq[String]: $seq")
}

import M._                                                        // ❹
m(List(1,2,3))
m(List("one", "two", "three"))
```

❶ 上面的代码中定义了两个具有特殊用途的隐式对象，这两个对象将用于解决由于类型擦除导致的方法二义性问题。

❷ 重新定义输入参数为 Seq[Int] 类型的方法。现在该方法新增了第二个参数列表，新增的参数列表希望能够接收到一个隐式 IntMarker 对象。请注意该对象的类型是 IntMarker.type。该类型引用了单例对象的类型！

❸ 重新定义输入参数为 Seq[String] 的方法。

❹ 导入并使用隐式值和方法。这些代码能够顺利通过编译并打印出正确的输出。

现在，即便发生了类型擦除，编译器还是会将这两个 m 方法视作不同的方法。你也许会思考为什么要发明新的类型，而不使用隐式 Int 和 String 值呢？我们并不推荐使用常用类型的隐式值。为什么呢？假设当前作用域中某一模块定义了 String 类型的一个隐式参数以及一个“默认”的隐式 String 类型值，之后该作用域的另外一个模块也定义了隐式 String 参数，那么这两个隐式参数便会导致系统出错。首先，假设第二个模块并未定义“默认”隐式值，而希望用户自己能够定义适用于应用程序的隐式值。如果用户没有定义该隐式值，那么该模块便会使用其他模块的隐式值，这可能会导致无法预期的行为。如果用户定义了隐式值，那么这两个出现在相同作用域中的隐式值将会导致二义性，而编译器就会抛出错误。

第一种场景会导致无法预期的行为发生，而第二类场景则会立即触发错误。

比较安全的做法是减少使用隐式参数及隐式值，改用那些专为此目的设计的特有类型。

> 尽量避免对 Int 和 String 那样的常用类型使用隐式参数及其对应值。因为与其他类型相比，这些类型更有可能会在多处定义其对应的隐式对象。假如这些隐式定义被导入到相同的作用域内，便会导致程序 bug 或编译错误。

我们会在第 14 章中更详细地讨论类型擦除的相关内容。

## 5.2.6 改善报错信息

我们暂时先回到集合 API 和 CanBuildFrom 构造器的相关示例中。如果试图对某一个未定义对应 CanBuildFrom 的目标类型调用 map 方法，那会发生什么呢？

```
scala> case class ListWrapper(list: List[Int])
defined class ListWrapper

scala> List(1,2,3).map[Int,ListWrapper](_ * 2)
<console>:10: error: Cannot construct a collection of type ListWrapper
with elements of type Int based on a collection of type List[Int].
              List(1,2,3).map[Int,ListWrapper](_*2)
                                               ^
```

上述代码在 map 方法中指定了明确的类型注解 map[Int,ListWrapper]，这确保了输出的对象类型是我们所希望的 ListWrapper 类型，而不是默认的 List[Int] 类型。而该注解同时也触发了我们预期引发的错误。请注意描述性的错误信息：Cannot construct a collection of type ListWrapper with elements of type Int based on a collection of type List[Int]（如果输入类型为 List[Int]，我们无法通过类型是 Int 类型的元素构造类型为 ListWrapper 的集合）。这并不是通常情况下编译器因为无法找到某一隐式参数的隐式值时所产生的默认信息。实际上，在声明 CanBuildFrom 时指定了一个名为 scala.annotation.implicitNotFound 的注解（annotation，与 Java 的 annotation 相似，http://www.scala-lang.org/api/current/scala/annotation/implicitNotFound.html）。该注解指定了那些错误信息的格式字符串（如果想了解更多 Scala 注解相关的内容，请查阅 23.2 节）。CanBuildFrom 的声明体如下所示：

```
@implicitNotFound(msg =
  "Cannot construct a collection of type ${To} with elements of type ${Elem}" +
  " based on a collection of type ${From}.")
trait CanBuildFrom[-From, -Elem, +To] {...}
```

你只能对那些专为满足隐式参数而定义的、用作隐式值的类型使用这类注解。而这些注解无法用于像 SRow.get[T] 那样的接受隐式参数输入的方法之上。

这也解释了为什么构建隐式时应该使用自定义类型，而不应使用更为常见的类型，比如说 Int 类型、String 类型甚至是像 SRow 示例中出现的（JRow, String）=> T 函数类型。使用自定义类型，你能为用户提供更有用的错误信息。

## 5.2.7 虚类型

我们之前已经学习了像 CanBuildFrom 那种可以添加行为的隐式参数。也使用了已经定义好的可作为 API 调用时的隐式值，如：作用于集合的 toMap 方法和 <:< 类型的隐式实例。

接下来我们要进行的步骤是移除所有的实例，仅仅留下需要的类型。这类定义好的没有任何实例的类型被称为*虚类型*（phantom type）。尽管虚类型的名字听起来较为花哨，不过它仅表明我们只关心类型本身。虚函数便是作为这样一个标志而存在的，表明我们不会使用

该类型的任何实例。

尽管我们即将谈论的虚类型与隐式没有任何关系，不过它却能用在我们之前解决的设计问题上。

对于定义必须按照某一特定顺序执行的工作流而言，虚类型作用很大。我们举一个简化版的工资计算器的例子。根据美国税法，在计算工资税之前，保险基金以及某些退休存款（401k）账户可以作为抵税项先从工资中扣除。因此，工资计算器必须首先执行“扣税前”的减项操作，然后进行扣税，最后计算器会扣除扣税后的其他减项并算出净收入。

下面便是该示例可能的一种实现：

```
// src/main/scala/progscala2/implicits/phantom-types.scala

// 工资计算的工作流程。

package progscala.implicits.payroll

sealed trait PreTaxDeductions
sealed trait PostTaxDeductions
sealed trait Final

// 为了简单起见,此处使用Float类型表示金额。不推荐大家这样做……
case class Employee(
  name: String,
  annualSalary: Float,
  taxRate: Float,  // 为了简化起见，所有的税种税率相同。
  insurancePremiumsPerPayPeriod: Float,
  _401kDeductionRate: Float,  // 税前扣除项,美国退休储蓄计划扣款。
  postTaxDeductions: Float)

case class Pay[Step](employee: Employee, netPay: Float)

object Payroll {
  // 每两周发一次薪水。为了简化问题,我们认定每年正好有52周。
  def start(employee: Employee): Pay[PreTaxDeductions] =
    Pay[PreTaxDeductions](employee, employee.annualSalary / 26.0F)

  def minusInsurance(pay: Pay[PreTaxDeductions]): Pay[PreTaxDeductions] = {
    val newNet = pay.netPay - pay.employee.insurancePremiumsPerPayPeriod
    pay copy (netPay = newNet)
  }

  def minus401k(pay: Pay[PreTaxDeductions]): Pay[PreTaxDeductions] = {
    val newNet = pay.netPay - (pay.employee._401kDeductionRate * pay.netPay)
    pay copy (netPay = newNet)
  }

  def minusTax(pay: Pay[PreTaxDeductions]): Pay[PostTaxDeductions] = {
    val newNet = pay.netPay - (pay.employee.taxRate * pay.netPay)
    pay copy (netPay = newNet)
  }

  def minusFinalDeductions(pay: Pay[PostTaxDeductions]): Pay[Final] = {
```

```
    val newNet = pay.netPay - pay.employee.postTaxDeductions
    pay copy (netPay = newNet)
  }
}

object CalculatePayroll {
  def main(args: Array[String]) = {
    val e = Employee("Buck Trends", 100000.0F, 0.25F, 200F, 0.10F, 0.05F)
    val pay1 = Payroll start e
    // 401K和保险扣除的顺序可以交换
    val pay2 = Payroll minus401k pay1
    val pay3 = Payroll minusInsurance pay2
    val pay4 = Payroll minusTax pay3
    val pay  = Payroll minusFinalDeductions pay4
    val twoWeekGross = e.annualSalary / 26.0F
    val twoWeekNet   = pay.netPay
    val percent      = (twoWeekNet / twoWeekGross) * 100
    println(s"For ${e.name}, the gross vs. net pay every 2 weeks is:")
    println(
      f"  $$${twoWeekGross}%.2f vs. $$${twoWeekNet}%.2f or ${percent}%.1f%%")
  }
}
```

代码已经通过了 sbt 的编译，因此可以在 sbt 命令行下执行程序：

```
> run-main progscala.implicits.payroll.CalculatePayroll
[info] Running progscala.implicits.payroll.CalculatePayroll
For Buck Trends, the gross vs. net pay every 2 weeks is:
  $3846.15 vs. $2446.10 or 63.6%
```

请注意，这些密封特征（sealed trait）本身不包含任何数据，而且没有实现这些 trait 的类。由于这些 trait 被“密封”了，我们无法在其他文件中实现这些 trait，因此，这些 trait 只能起到标志的作用。

Pay 类型将这些标志用作类型参数，而这些参数也被当作标记，调用 Payroll 类的每一个方法均需要传入 Pay[Step] 对象，该对象包含的 Step 参数表示了当前执行的步骤。这也是我们在调用 minus401k 方法时无法传入 Pay[PostTaxDeductions] 对象的原因。因此，这些 API 能够确保税务规定的执行。

CalculatePayroll 对象演示了这些 API 的使用方法，假如你尝试修改 API 的调用顺序，如在调用 Payroll.minusTax 方法之前调用 Payroll.minusFinalDeductions 方法，那么编译将会抛出错误。

请注意，本示例结尾处调用的 println 方法使用了可插入字符串，该字符串与我们之前在 3.13 节讨论的示例非常相近。

事实上，上述代码中的 main 函数不是十分简洁，这种复杂性也破坏了代码的美感。为了解决这个问题，我们引入了微软的函数式编程语言——F# 语言中的“管道”操作符。下面的示例代码改编自 James Iry 的一篇博文（http://james-iry.blogspot.ch/2010/10/phantom-types-in-haskell-and-scala.html）。

```
// src/main/scala/progscala2/implicits/phantom-types-pipeline.scala
package progscala.implicits.payroll
import scala.language.implicitConversions

object Pipeline {
  implicit class toPiped[V](value:V) {
    def |>[R] (f : V => R) = f(value)
  }
}

object CalculatePayroll2 {
  def main(args: Array[String]) = {
    import Pipeline._
    import Payroll._

    val e = Employee("Buck Trends", 100000.0F, 0.25F, 200F, 0.10F, 0.05F)
    val pay = start(e) |>
      minus401k        |>
      minusInsurance   |>
      minusTax         |>
      minusFinalDeductions
    val twoWeekGross = e.annualSalary / 26.0F
    val twoWeekNet   = pay.netPay
    val percent      = (twoWeekNet / twoWeekGross) * 100
    println(s"For ${e.name}, the gross vs. net pay every 2 weeks is:")
    println(
      f"  $$${twoWeekGross}%.2f vs. $$${twoWeekNet}%.2f or ${percent}%.1f%%")
  }
}
```

main 方法中包含了更为简洁的一组操作步骤，这些操作均调用了 Payroll 方法。值得注意的是，尽管管道操作符 |> 看上去很酷，但它其实只是重排了表达式中各个标记的次序。例如：|> 操作符对下列表达式就进行了转化：

```
pay1 |> Payroll.minus401k
```

转化之后形成了下列表达式：

```
Payroll.minus401k(pay1)
```

### 5.2.8 隐式参数遵循的规则

回顾下隐式参数，下面的编注栏中列出了隐式参数应遵循的通用规则。

**隐式参数所遵循的规则**

(1) 只有最后一个参数列表中允许出现隐式参数，这也适用于只有一个参数列表的情况。

(2) implicit 关键字必须出现在参数列表的最左边，而且只能出现一次。列表中出现在 implicit 关键字之后的参数都不是“非隐式”的。

(3) 假如参数列表以 implicit 关键字开头，那么所有的参数都是隐式的。

请留意下面示例中出现的错误信息，这些示例均违背了上述规则：

```
scala> class Bad1 {
     |   def m(i: Int, implicit s: String) = "boo"
<console>:2: error: identifier expected but 'implicit' found.
         def m(i: Int, implicit s: String) = "boo"
                       ^

scala> }
<console>:1: error: eof expected but '}' found.
       }
       ^

scala> class Bad2 {
     |   def m(i: Int)(implicit s: String)(implicit d: Double) = "boo"
<console>:2: error: '=' expected but '(' found.
         def m(i: Int)(implicit s: String)(implicit d: Double) = "boo"
                                          ^

scala> }
<console>:1: error: eof expected but '}' found.
       }
       ^

scala> class Good1 {
     |   def m(i: Int)(implicit s: String, d: Double) = "boo"
     | }
defined class Good1

scala> class Good2 {
     |   def m(implicit i: Int, s: String, d: Double) = "boo"
     | }
defined class Good2
```

## 5.3 隐式转换

在 2.8.8 节，我们学习了创建 pair 的一些方法：

```
(1, "one")
1 -> "one"
1 → "one"         // 用 → 替代 ->
Tuple2(1, "one")
Pair(1, "one")    // Scala 2.11不推荐使用这种写法
```

本文不会对 `(a, b)` 和 `a -> b` 这两类字面量格式的 Scala 语法进行讲解。

在初始化 `Map` 对象时，我们经常使用 `a -> b`（或等价的 `a → b`）这种写法：

```
scala> Map("one" -> 1, "two" -> 2)
res0: scala.collection.immutable.Map[String,Int] = Map(one -> 1, two -> 2)
```

上述代码中调用了 `Map.apply` 方法，而该方法的输入是一组可变数量的 pair 对象：

```
def apply[A, B](elems: (A, B)*): Map[A, B]
```

事实上，Scala 根本不知道 a -> b 意味着什么，因此上述方法并非没有意义。这种“字面量”格式实际上运用了方法 -> 和一个特殊的 Scala 特性——隐式转换。通过运用隐式转换，我们可以在任意两种类型值之间插入函数 ->。与此同时，由于 a -> b 并不是元组的字面量语法，因此 Scala 必须通过某些方式将该表达式转化为元组 (a，b)。

很明显，我们首先需要定义方法 ->，但是应该在哪儿定义该方法呢？ 我们希望该方法能够适用于各种可能的元组首元素的类型。即便我们能够对所有需要添加该方法的类型进行编辑，我们也不应该这样做，编辑所有的类型既不实际也不明智。

定义该方法的技巧是使用一个具有 -> 方法的“封装”对象，Scala 已经在 Predef 对象中定义了该对象：

```
implicit final class ArrowAssoc[A](val self: A) {
  def -> [B](y: B): Tuple2[A, B] = Tuple2(self, y)
}
```

（为了更清楚地说明问题，我在此处省略了实际声明体中的一些不重要的细节）。与 Java 类似，此处的 final 关键字表明了我们无法创建 ArrowAssoc 的子类声明。

我们可以依照下列代码构造 Map 对象：

```
scala> Map(new ArrowAssoc("one") -> 1, new ArrowAssoc("two") -> 2)
res0: scala.collection.immutable.Map[String,Int] = Map(one -> 1, two -> 2)
```

这种写法在简洁程度上不如直接使用 Map( ("one", 1), ("two", 2) )。不过，这种写法能实现我们想要的部分功能。ArrowAssoc 类能够接受任何类型的对象，当执行 -> 方法时，将返回一个 pair 对象。

在这个示例中，implicit 关键字又一次起作用了。由于 ArrowAssoc 类被声明为 implicit 类，因此编译器将执行下列逻辑。

(1) 编译器发现我们试图对 String 对象执行 -> 方法（例如“one” -> 1）。
(2) 由于 String 未定义 -> 方法，编译器将检查当前作用域中是否存在定义了该方法的隐式转换。
(3) 编译器发现了 ArrowAssoc 类。
(4) 编译器将创建 ArrowAssoc 对象，并向其传入 one 字符串。
(5) 之后，编译器将解析表达式中的 -> 1 部分代码，并确认了整个表达式的类型与 Map.apply 方法的预期类型相吻合，即两者均为 pair 实例。

如果希望执行隐式转换，那么在声明时必须使用 implicit 关键字，能够执行隐式转换的无外乎两类：构造方法中只接受单一参数的类型或者是只接受单一参数的方法。

在 Scala 2.10 诞生之前，如果想要实现隐式转换，就必须定义一个不含 implicit 关键字的封装类，并且要在封装类中定义一个将执行转换的隐式方法。由于这种两段式的定义方式毫无意义，因此现在我们可以直接将该类标注为隐式类并清除该方法。例如，ArrowAssoc 看上去就好像 Scala 2.10 版本之前的封装类（any2ArrowAssoc 方法仍然存在，不过现在已经不建议使用了）。

```
final class ArrowAssoc[A](val self: A) {
```

```
  def -> [B](y: B): Tuple2[A, B] = Tuple2(self, y)
}
...
implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A] = new ArrowAssoc(x)
```

尽管隐式方法仍然被使用，但是我们现在只有在将某类型转换为另一个已经存在的类型时才会使用这些方法，而且这些隐式方法并未使用 implicit 关键字进行声明。

我们之前在讲解*虚类型*（phantom type）时曾定义了管道操作，这也是隐式转换的又一示例：

```
... pay1 |> Payroll.minus401k ...
```

滥用隐式方法会导致难以调试的诡异行为。因此从 Scala 2.10 开始，隐式方法变成了 Scala 的可选特性。假如你希望使用这一特性，你应该通过 import 语句 import scala.langage.implicitConversions 开启这一特性，你也可以使用全局的编译器选项 -language:implictConversions 开启该特性。

以下是编译器进行查找和使用转换方法时的查询规则。

(1) 假如调用的对象和方法成功通过了组合类型检查，那么类型转换不会被执行。
(2) 编译器只会考虑使用了 implicit 关键字的类和方法。
(3) 编译器只会考虑当前作用域内的隐式类，隐式方法，以及目标类型的伴生对象中定义的隐式方法（本文后续部分将讲讨论这种情况）。
(4) 隐式方法无法串行处理，我们无法通过一个中间类型，使用串行的隐式方法将起始类型转换成目标类型。编译器执行隐式转换时只会考虑那些接受单一类型实例输入且返回目标类型实例的方法。
(5) 假如当前适用多条转换方法，那么将不会执行转换操作。编译器要求有且必须只有一条满足条件的隐式方法，以免产生二义性。

规则 3 中提到了伴生对象中定义的隐式方法，该规则具有以下含义。首先，假如隐式转换出现在当前作用域之外，那么便不会执行该隐式转换。假如在当前作用域内，这意味着隐式转换是在封闭作用域内声明的，或者当前作用域已经导入了隐式转换所在的其他作用域，比方说导入了其他作用域中某一个定义了一些隐式转换的对象。

不过当需要执行转换时，假如转换的目标类型的伴生对象中定义了转换方法，那么编译器会自动导入伴生对象作用域，并最后查找该作用域。参见下面的示例：

```
// src/main/scala/progscala2/implicits/implicit-conversions-resolution2.sc
import scala.language.implicitConversions

case class Foo(s: String)
object Foo {
  implicit def fromString(s: String): Foo = Foo(s)
}

class O {
  def m1(foo: Foo) = println(foo)
  def m(s: String) = m1(s)
}
```

Foo 类型的伴生对象定义了基于 String 类型的转换。而 O.m 方法试图在调用 O.m1 方法时传入 String 类型，但 O.m1 方法却期望输入 Foo 对象。尽管我们并未明确地将 Foo 伴生对象导入当前的作用域，编译器还是能够发现 Foo.fromString 转换方法的存在。

不过，假如当前作用域中存在另外一个 Foo 转换方法，该转换方法将会覆盖 Foo.fromString 转换：

```
// src/main/scala/progscala2/implicits/implicit-conversions-resolution.sc
import scala.language.implicitConversions

case class Foo(s: String)
object Foo {
  implicit def fromString(s: String): Foo = Foo(s)
}

class O {
  def m1(foo: Foo) = println(foo)
  def m(s: String) = m1(s)
}
```

现在，编译器将使用覆盖后的转换方法。

我们之前提到过，除了那些在伴生对象中定义的隐式值和转换方法之外，我们推荐将隐式值和转换方法放到一个名为 implicits 的特殊包或名为 Implicits 的对象中。这样做的好处是：读者能更清楚地知道是哪个 import 语句导入了代码使用的自定义隐式。

最后，请注意 Scala 为像 String 和 Array 这样的 Java 类型提供了一些隐式封装类型。例如：下面代码中出现的方法看上去像是 String 类方法，但是这些方法实际上是由 WrappedString 类型（http://www.scala-lang.org/api/current/#scala.collection.immutable.WrappedString）实现的：

```
scala> val s = "Programming Scala"
s: String = Programming Scala

scala> s.reverse
res0: String = alacS gnimmargorP

scala> s.capitalize
res1: String = Programming Scala

scala> s.foreach(c => print(s"$c-"))
P-r-o-g-r-a-m-m-i-n-g- -S-c-a-l-a-
```

在 Scala 自带的“封装”类型中定义的隐式转换会一直出现在当前作用域中。这些隐式转换更确切的讲被定义在 Predef 中。

我们之前用过的 Range 类型也是一类“封装”类型。例如：代码 1 to 100 by 3 代表从 1 到 100 每隔 3 个数取一个整数。你现在应该能猜测出 to、by 这样的字样以及 scala 独有的 util 单词都是封装对象中的方法，它们并不是 Scala 关键字。例如，scala.runtime.RichInt（http://www.scala-lang.org/api/current/scala/runtime/RichInt.html）封装了 Int 类型，它包含这些方法。Scala 在同一个包中定义了适用于其他数值类型的类似的封装类型：RichLong、RichFloat、RichDouble

和 RichChar。而 scala.math.BigInt（http://www.scala-lang.org/api/current/scala/math/BigInt.html）和 scala.math.BigDecimal（http://www.scala-lang.org/api/current/scala/math/BigDecimal.html）它们本身便是 Java 同名类的封装类型，因此它们没有自己的封装类型。它们自己本身便可以实现 to、until 和 by 方法。

### 5.3.1　构建独有的字符串插入器

我们再来学习最后一个隐式转换的示例，该示例允许我们通过隐式转换定义我们自己独有的字符串插入器。我们回顾一下之前在 3.13 节学到的内容：Scala 提供了一些内置的方法通过插入方式对字符串进行格式化，例如：

```
val name = ("Buck", "Trends")
println(s"Hello, ${name._1} ${name._2}")
```

当编译器看到像 x"foo bar" 这样的表达式时，它会查找 scala.StringContext（http://www.scala-lang.org/api/current/scala/StringContext.html）中定义的 x 方法。上述代码的最后一行可以被转换成：

```
StringContext("Hello, ", " ", "").s(name._1, name._2)
```

传递给 StringContext.apply 方法的参数其实是 ${...} 表达式中抽取出的各个部分。传递给 s 的参数则是抽取后的表达式。（请提供变量值，对上述代码进行试验）StringContext 中还定义了 f 和 raw 方法，这些方法能够帮助 f 格式以及 raw 格式的插入器工作。

通过使用隐式转换我们可以为 StringContext 添加新的方法，对其进行“扩展”，进而定义属于自己的插入器。我们对 StringContext Scaladoc 页面（http://www.scala-lang.org/api/current/scala/StringContext.html）中出现的示例进行扩充，编写一个能够将简单的 JSON 字符串转化为 scala.util.parsing.json.JSONObject 对象（http://www.scala-lang.org/api/current/scala-parser-combinators/#scala.util.parsing.json.JSONObject）的插入器。[1]

为了简化，我们会做一些假设。首先，我们不会对数组或嵌套对象进行处理，只处理像 {"a": "A", "b": 123, "c": 3.14159} 这样的扁平的 JSON 表达式。其次，我们要求 JSON 的 key 都是固定值，而 value 值则是指定的可插入参数，例如：{"a": $a, "b": $b, "c": $c}。假如我们需要使用该插入器生成 value 值不同但结构相似的 JSON 对象，第二条限制便是合理的。插入器实现代码如下：

```
// src/main/scala/progscala2/implicits/custom-string-interpolator.sc
import scala.util.parsing.json._

object Interpolators {
  implicit class jsonForStringContext(val sc: StringContext) {      // ❶
    def json(values: Any*): JSONObject = {                          // ❷
      val keyRE = """^[\s{,]*(\S+):\s*""".r                         // ❸
      val keys = sc.parts map {                                     // ❹
        case keyRE(key) => key
```

注 1：Scala 2.11 中，json 包被放置到一个独立的解析器－组合器（parser-combinator）库中，我们将在 20.1 节对其进行讲解。

```
        case str => str
      }
      val kvs = keys zip values                                  // ❺
      JSONObject(kvs.toMap)                                      // ❻
    }
  }
}

import Interpolators._

val name = "Dean Wampler"
val book = "Programming Scala, Second Edition"

val jsonobj = json"{name: $name, book: $book}"                   // ❼
println(jsonobj)
```

❶ 为了限定隐式类的作用域，必须在对象内定义隐式类（出于安全的考虑）。类定义之后出现的 `import` 语句将该实现类导入到代码所在的作用域中。

❷ `json` 方法。该方法的输入是字符串中嵌套的参数，该方法返回构造好的 `scala.util.parsing.json.JSONObject` 对　象（http://www.scala-lang.org/api/current/scala-parser-combinators/#scala.util.parsing.json.JSONObject）。

❸ 用于从字符串片段中抽取 key 名称的正则表达式。

❹ 使用正则表达式从字符片段各个部分中抽取出 key 的名称。假如正则表达式不匹配，那么将使用整个字符串作为 key 值。不过在这种情况下抛出错误可能是一个更好的选择，因为这样便能避免出现无效的 key 字符串。

❺ 将 key 值和 value 值一并“压缩”成健－值对集合。我们在解释完代码之后会对 `zip` 方法进行讨论。

❻ 使用健值对构建 `Map` 对象，并使用 `Map` 对象构造 `JsonObject` 对象。

❼ 使用我们自己的字符串插入器，使用起来与内置插入器无异。

自定义的字符串插入器无需像 `s`、`f` 和 `raw` 插入器那样返回 `String` 类型。我们会返回 `JSONObject` 类型。因此，自定义字符串插入器可以用作由字符串中封装的数据所驱动的实例工厂。

就像拉链一样，集合中的 `zip` 方法能很方便地将两个集合中的值缝合在一起。如下所示：

```
scala> val keys = List("a", "b", "c", "d")
keys: List[String] = List(a, b, c, d)

scala> val values = List("A", 123, 3.14159)
values: List[Any] = List(A, 123, 3.14159)

scala> val keysValues = keys zip values
keysValues: List[(String, Any)] = List((a,A), (b,123), (c,3.14159))
```

合并后的集合中的元素类型为二元元祖（如 `key1`、`value1` 等）。需要注意的是，由于某一列表比另一个列表长，因此列表尾部多余的元素便会被丢弃掉。JSON 字符串中字符串片段数量比值参数的数量多一个，但我们实际上希望出现这类行为，因为我们并不需要字符

串尾部的最后一个片段。

下面列出了在 REPL 会话中运行示例代码后，最后两行中的输出：

```
scala> val jsonobj = json"{name: $name, book: $book}"
jsonobj: scala.util.parsing.json.JSONObject = \
  {"name" : "Dean Wampler", "book" : "Programming Scala, Second Edition"}

scala> println(jsonobj)
{"name" : "Dean Wampler", "book" : "Programming Scala, Second Edition"}
```

## 5.3.2 表达式问题

我们再思考一下之前已经解决过的问题：我们已经能够在不修改任何相关源代码的情况下成功地为所有类型添加新的方法。

在不修改源代码的情况下扩展模块的期望被称为*表达式问题*（expression problem）。这一概念是由 Philip Wadler（http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt）创造的。

面向对象编程通过*子类型化*（subtyping）解决了这一问题，更精确地讲是*子类型多态*（subtype polymorphism）。当需要对某些行为进行修改时，我们会首先编写抽象体，之后使用抽象体的继承类修改行为。Bertrand Meyer 提出的*开/闭准则*（open/closed principle）对 OOP 这一方法进行了描述，在这一准则中，基类声明了抽象的行为，而子类型则在不修改基类型的情况下对抽象行为进行适当的修改。

Scala 当然也支持这一技术，不过这项技术是有弊端的。我们是否应该在类继承关系中的父类型中定义这些行为呢？假如只有少数几种情况下我们才需要这种行为，而大多数情况下该行为只是客户端代码加载时的负担？

这种方法有可能会成为负担。首先，这些额外的无用代码会占据系统资源。尽管有些应用并不会为此影响，但有些成熟代码会不可避免地出现很多损耗。其次，我们为此需要对大多数定义的行为进行一次次的修改。一旦对这些无用行为进行修改，即便不需要使用这一行为的客户端代码也需要进行多余的修改。

这一问题引发了*单一职责原则*（single responsibility principle）这一经典设计原则。该原则鼓励我们在定义抽象以及实现体时，只提供某一单一的行为。

在实际场合中，有时我们需要某一对象来把其他行为组合起来。例如，服务对象往往需要"混入"日志信息这一功能。正如我们在 3.14 节中展示的那样，Scala 很容易实现这些混入 trait。我们甚至可以在声明对象时为其指定 trait。

动态类型语言通常都提供*元编程*（metaprogramming）功能，该功能允许用户在元编程运行的环境下，不必修改源代码就可以修改类。对于类导致的问题，尤其是这些类定义了几乎不被使用的行为，这一方法具有一定的效果。不幸的是，对于大多数动态语言而言，运行时对类型做的任一修改都会作用于全局，因此全局内的所有用户都会被影响。

通过使用 Scala 的隐式转换特征，我们可以得到另外一种通过静态类型实现元编程的方法，我们称之为*类型类*（type class）。Haskell 率先提出这一概念，可以参考文章"A Gentle

Introduction to Haskell”（http://www.haskell.org/tutorial/classes.html）。类型类这一名称源于 Haskell，请不要将其与 Scala 常用类所混淆（如果某对象属于某类型类，那么它必须实现类型类所定义的行为，与通常意义上的类相比，类型类更接近于接口）。

## 5.4 类型类模式

由于我们可以即兴地为类型添加行为，因此使用类型类可以帮助我们避免创建像 Java 中 `Object` 那样的抽象体，例如“厨房水槽”这样的抽象体。Scala 的 pair 构造语法 -> 便很好地说明了这点。回想一下，我们并没有修改这些类型；我们只是通过隐式机制将对象封装到某个提供了我们所需行为的类型中。修改之后的结果与我们直接修改类型源代码的结果一样。

Scala 作为一门 JVM 语言，也继承了 Java 中无所不在的 `Object.toString` 方法。Java 默认的 `toString` 方法只会显示类型名称和对象在 JVM 堆中的地址，因此它本身并没有多大的价值。而 Scala 在 case 类中使用的语法则不同，该语法更加有用，也更具可读性。有时候打印出像 JSON 或 XML 样的具有机器可读性的格式是很有价值的。通过使用隐式转换，我们可以为任何类型添加 `toJSON` 和 `toXML` 方法。如果对象中未定义 `toString` 方法，我们也能通过隐式转换定义该方法。

Haskell 语言中的类型类能定义等价于接口的类型，之后我们便能实现各种具体类型。Scala 中的*类型类模式*（type class pattern）新增了接口部分，这一功能是之前的隐式转换示例所未提供的。

让我们阅读一下 `toJSON` 类型类的某一实现：

```
// src/main/scala/progscala2/implicits/toJSON-type-class.sc

case class Address(street: String, city: String)
case class Person(name: String, address: Address)

trait ToJSON {
  def toJSON(level: Int = 0): String

  val INDENTATION = "  "
  def indentation(level: Int = 0): (String,String) =
    (INDENTATION * level, INDENTATION * (level+1))
}

implicit class AddressToJSON(address: Address) extends ToJSON {
  def toJSON(level: Int = 0): String = {
    val (outdent, indent) = indentation(level)
    s"""{
      |${indent}"street": "${address.street}",
      |${indent}"city":   "${address.city}"
      |$outdent}""".stripMargin
  }
}

implicit class PersonToJSON(person: Person) extends ToJSON {
```

```
  def toJSON(level: Int = 0): String = {
    val (outdent, indent) = indentation(level)
    s"""{
      |${indent}"name":    "${person.name}",
      |${indent}"address": ${person.address.toJSON(level + 1)}
      |$outdent}""".stripMargin
  }
}

val a = Address("1 Scala Lane", "Anytown")
val p = Person("Buck Trends", a)

println(a.toJSON())
println()
println(p.toJSON())
```

为了简化起见，Person 和 Address 类型中只包含了少量的字段，同时我们希望将对象格式化为多行表示的 JSON 字符串，而不是 scala.util.parsing.json 包（http://www.scala-lang.org/api/current/scala-parser-combinators/#scala.util.parsing.json.package）里定义的对象类型。（请参考 20.1 节）。

我们在 ToJSON trait 中定义了默认的缩进字符串，也定义了用于计算字段实际缩进长度的方法以及 JSON 对象中包含的闭合括号 {...}。toJSON 方法的输入参数指定了当前的缩进级别；也就是说，缩进多少个 INDENTATION 单元。由于 toJSON 方法要求输入这一参数，客户程序就必须提供空括号或者其他缩进级别。需要注意的是输入的缩进级别是用双引号包裹的字符串值，而不是整数值。

上述脚本输出如下：

```
{
  "street": "1 Scala Lane",
  "city":   "Anytown"
}

{
  "name":    "Buck Trends",
  "address": {
    "street": "1 Scala Lane",
    "city":   "Anytown"
  }
}
```

Scala 不允许同时使用 implict 和 case 关键字。也就是说，隐式类不能同时是一个 case 类。由于 case 类不会执行通过隐式所自动生成的额外代码，因此隐式 case 类本身也就没有任何意义。可见隐式类的用途非常窄。

请注意，我们通过使用该机制为已经存在的类添加了方法。因此，*扩展方法*（extension method）是类型类的另一用途。在 C# 和 F# 这样的语言中也存在另一种扩展方法的机制（请查阅微软开发网络（https://msdn.microsoft.com/en-us/library/bb383977.aspx）上的关于扩展方法的相关页面）。尽管 C# 和 F# 提供的机制可能更加直白，但是类型类的应用却更为广泛。

另一方面，与面向对象继承关系中常出现的*子类型多态*（subtype polymorphism）不同，`toJSON` 方法所提供的多态行为并未绑定类型系统。因此类型类提供的这一功能也被称为*特设多态*（ad hoc polymorphism）。我们之前在 2.13 节中介绍过第三类多态：*参数化多态*（paremetric polymorphism）。在这一类多态中，像 `Seq[A]` 这样的容器的行为因 `A` 类型不同而不同。

我们常会有这样的错觉：如果定义了一组类，每个类只提供某一特定的行为，对于大多数客户而言，这些特定的行为并没有什么作用。而类型类模式最适合用于这种情况，对这组类应用类型类模式能够使客户从中获益。善用该模式能够在维护单一职责原则的同时平衡多个客户的需求。

## 5.5 隐式所导致的技术问题

那么隐式有哪些问题呢？在定义类型时，为什么不将类型定义为仅比字段包（字段包有时候又称为*贫血类型*，anemic type）稍丰富一点，只提供非常少的行为的类型呢？然后再使用类型类添加所有的行为呢？

首先，你需要花时间编写用于定义隐式的额外代码，而且编译器也必须费力处理隐式。因此，编译那些大量应用隐式的项目需要耗费很多的时间。

隐式转换同样会造成额外的运行开销，这是因为封装类型会引入额外的中间层。尽管编译器会内联某些方法调用，而且 JVM 也会执行一些额外的优化，但是这样还是会有些额外开销。我们会在第 14 章谈论*值类型*，用于在编译期间消除值类型的额外运行开销。

当隐式特征与其他 Scala 特征，尤其是子类型特征发生交集时，会产生一些技术问题（关于子类型特征的完整内容，请阅读 scala-debate email 组的讨论贴）。

我们用一个简单示例来说明这一点：

```
// src/main/scala/progscala2/implicits/type-classes-subtyping.sc

trait Stringizer[+T] {
  def stringize: String
}

implicit class AnyStringizer(a: Any) extends Stringizer[Any] {
  def stringize: String = a match {
    case s: String => s
    case i: Int => (i*10).toString
    case f: Float => (f*10.1).toString
    case other =>
      throw new UnsupportedOperationException(s"Can't stringize $other")
  }
}

val list: List[Any] = List(1, 2.2F, "three", 'symbol)

list foreach { (x:Any) =>
  try {
    println(s"$x: ${x.stringize}")
```

```
    } catch {
      case e: java.lang.UnsupportedOperationException => println(e)
    }
  }
```

我们定义了一个名为 Stringizer 的抽象体。如果按照之前 ToJSON 示例的做法，我们会为所有我们希望能字符串化的类型创建隐式类。这本身就是一个问题。如果我们希望处理一组不同的类型实例，我们只能在 list 类型的 map 方法内隐式地传入一个 Stringizer 实例。因此，我们就必须定义一个 AnyStringerize 类，该类知道如何对我们已知的所有类型进行处理。这些类型甚至还包含用于抛出异常的 default 子句。

这种实现方式非常不美观，同时也违背了面向对象编程中的一条核心规则——你不应该使用 switch 语句对可能发生变化的类型进行判断。相反，你应该利用多态分发任务，这类似于 toString 方法在 Scala 和 Java 语言中的运作方式。

如果你想更深入地了解 ToJSON 方法作用于一组对象的具体方法，请查看相关示例代码：implicits/type-classes-subtyping2.sc。

最后，我将列出帮助我们避免某些潜在问题的一些技巧。

无论何时都要为隐式转换方法指定返回类型。否则，类型推导推断出的返回类型可能会导致预料之外的结果。

另外，虽然编译器会执行一些“方便”用户的转换。但是目前来看这些转换带来的麻烦多过益处（以后推出的 Scala 也许会修改这些行为）。

首先，假如你为某一类型定义方法 +，并试图将该方法应用到某一不属于该类型的实例上，那么编译器会调用该实例的 toString 方法，这样一来便能执行 String 类型的 + 操作（合并字符串操作）。这可以解释某些特定情况下出现像 String 是错误类型的奇怪错误。

与此同时，如果有必要的话，编译器会将方法的输入参数自动组合成一个元组。有时候这一行为会给人造成困扰。幸运的是，Scala 2.11 现在会抛出警告信息。

```
scala> def m(pair:Tuple2[Int,String]) = println(pair)

scala> m(1, "two")
<console>:9: warning: Adapting argument list by creating a 2-tuple:
  this may not be what you want.
        signature: m(pair: (Int, String)): Unit
  given arguments: 1, "two"
 after adaptation: m((1, "two"): (Int, String))
              m(1,"two")
               ^
(1,two)
```

也许这个时候你已经有点晕乎了，不过这也没关系。我希望你能清楚认识到隐式是 Scala 中的一门强大的工具，而且在使用隐式时也能保持头脑清醒。

## 5.6 隐式解析规则

Scala 查找隐式时，会遵照一组复杂的搜寻规则，其中某些规则的设计初衷是为了解决潜在的二义性。[2]

尽管根据隐式所处的情景不同，会存在隐式方法、隐式值和隐式类这三类隐式。在下面的讨论中，我将使用“值”一词来表示隐式。

- Scala 会解析无须输入前缀路径的类型兼容隐式值。换句话说，隐式值定义在相同作用域中。例如：隐式值定义在相同代码块中，隐式值定义在相同类型中，隐式值定义在伴生对象中（如果存在的话），或者定义在父类型中。
- Scala 会解析那些导入到当前作用域的隐式值（这也无须输入前缀路径）。

第二条规则中提到的导入隐式值，其优先级高于已经在当前作用域的隐式值。

有的时候，会存在多个可能的类型兼容的匹配，这时 Scala 将挑选匹配度最高的隐式。举个例子，如果隐式参数类型是 `Foo` 类型，而当前作用域中既存在 `Foo` 类型的隐式值又存在 `AnyRef` 类型的隐式值，那么 Scala 会挑选类型为 `Foo` 的隐式值。

如果两个或多个隐式值可能引发歧义（例如：它们具有相同的类型），编译错误会被触发。

Scala 库中定义的隐式常常会被编译器加载到作用域中，而其他语言库中定义的隐式则需要通过 `import` 语句来加载进作用域。接下来我们将讨论这些被编译器加载的隐式。

## 5.7 Scala 内置的各种隐式

Scala 2.11 版库源代码中定义了超过 300 个隐式方法、隐式值和隐式类型。它们当中的大多数都是隐式方法，而隐式方法中的多数则被用于将某一类型转换成另一类型。掌握什么样的隐式可能会对代码产生影响对我们来说比较重要，因此我们会以小组的形式对这些隐式进行讨论，而不会是一一列举各个隐式。掌握每个隐式的详细内容绝非是最重要的，让读者能够“感觉”到隐式的类型才是最有益的。如果希望具备这一“直觉”，请概览本节。

我们之前已经讨论过作用于集合的 `CanBuildFrom` 构建器。现在有一个与它比较相似的 `CanCombineFrom` 构建器，与其名字意思相符 `CanCombineForm` 构建器被许多操作用于组合实例。但是本节不会列出这些构造器的定义。

`AnyVal` 类型的所有伴生对象均提供了丰富的转换方法，例如：将 `Int` 值转换为 `Long` 值。大多数时候你只需要调用该类型的 `toX` 方法，如下所示：

```
object Int {
  ...
  implicit def int2long(x: Int): Long = x.toLong
  ...
}
```

下列代码片段列举了一些 `AnyVal` 类型的隐式转换。需要注意的是由于 Scala 提供了隐式转

注 2：《Scala 语言规范》对这些规则给出了详细的定义。

换的功能，因此 Scala 无须像其他语言一样必须实现大多数常见的类型转换。

下列代码片段取自于 Byte（http://www.scala-lang.org/api/current/scala/Byte$.html）伴生对象：

```
implicit def byte2short(x: Byte): Short = x.toShort
implicit def byte2int(x: Byte): Int = x.toInt
implicit def byte2long(x: Byte): Long = x.toLong
implicit def byte2float(x: Byte): Float = x.toFloat
implicit def byte2double(x: Byte): Double = x.toDouble
```

下列代码片段取自 Char（http://www.scala-lang.org/api/current/scala/Char$.html）伴生对象：

```
// 下列代码片段取自Char伴生对象:
implicit def char2int(x: Char): Int = x.toInt
implicit def char2long(x: Char): Long = x.toLong
implicit def char2float(x: Char): Float = x.toFloat
implicit def char2double(x: Char): Double = x.toDouble
```

下列代码片段取自 Short（http://www.scala-lang.org/api/current/scala/Short$.html）伴生对象：

```
implicit def short2int(x: Short): Int = x.toInt
implicit def short2long(x: Short): Long = x.toLong
implicit def short2float(x: Short): Float = x.toFloat
implicit def short2double(x: Short): Double = x.toDouble
```

下列代码片段取自 Int（http://www.scala-lang.org/api/current/scala/Int$.html）伴生对象：

```
implicit def int2long(x: Int): Long = x.toLong
implicit def int2float(x: Int): Float = x.toFloat
implicit def int2double(x: Int): Double = x.toDouble
```

下列代码片段取自 Long（http://www.scala-lang.org/api/current/scala/Long$.html）伴生对象：

```
implicit def long2float(x: Long): Float = x.toFloat
implicit def long2double(x: Long): Double = x.toDouble
```

下列代码片段取自 Float（http://www.scala-lang.org/api/current/scala/Float$.html）伴生对象：

```
implicit def float2double(x: Float): Double = x.toDouble
```

BigInt 和 BigDecimal 类定义在 scala.math 包中，它们可以转换来自 AnyVal 类型和 Java 中对应的实现类型。下列代码片段取自于 BigDecimal（http://www.scala-lang.org/api/current/scala/math/BigDecimal$.html）伴生对象：

```
implicit def int2bigDecimal(i: Int): BigDecimal = apply(i)
implicit def long2bigDecimal(l: Long): BigDecimal = apply(l)
implicit def double2bigDecimal(d: Double): BigDecimal = ...
implicit def javaBigDecimal2bigDecimal(x: BigDec): BigDecimal = apply(x)
```

调用 apply 方法时调用了 BigDecimal 伴生对象的 apply 工厂方法。这些隐式提供了另一种简便的方式调用这些方法。

下列代码片段取自 BigInt（http://www.scala-lang.org/api/current/scala/math/BigInt$.html）伴生对象：

```
implicit def int2bigInt(i: Int): BigInt = apply(i)
implicit def long2bigInt(l: Long): BigInt = apply(l)
implicit def javaBigInteger2bigInt(x: BigInteger): BigInt = apply(x)
```

Option 对象可以转换成包含 0 个或 1 个元素的列表：

```
implicit def option2Iterable[A](xo: Option[A]): Iterable[A] = xo.toList
```

Scala 使用了一些 Java 中的类型，像 Array[T] 和 String 类型。和它们对应的 ArrayOps[T] 和 StringOps 类型向所有的 Scala 集合提供常用的方法。因此，无论是转换成还是转换自这些封装类型的隐式转换都是非常有用的。其他的转换函数则定义在名称中包含 Wrapper 单词的类型中。

Predef 中定义了大多数的隐式定义。其中的一些隐式定义包含了 @inline 标注（http://www.scala-lang.org/api/current/scala/inline.html），这一标注鼓励编译器努力尝试将函数调用内联（inline），以减少栈帧开销。与之对应的是 @noinline 标注（http://www.scala-lang.org/api/current/scala/noinline.html），该标注阻止编译器将方法调用内联化，即便是条件允许的情况。

一些方法能将某一类型转化为另一类型，例如：将某一类型封装成新的类型后，新的类型会提供新的方法：

```
@inline implicit def augmentString(x: String): StringOps = new StringOps(x)
@inline implicit def unaugmentString(x: StringOps): String = x.repr
implicit def tuple2ToZippedOps[T1, T2](x: (T1, T2))
  = new runtime.Tuple2Zipped.Ops(x)
implicit def tuple3ToZippedOps[T1, T2, T3](x: (T1, T2, T3))
  = new runtime.Tuple3Zipped.Ops(x)
implicit def genericArrayOps[T](xs: Array[T]): ArrayOps[T] = ...

implicit def booleanArrayOps(xs: Array[Boolean]): ArrayOps[Boolean] =
  = new ArrayOps.ofBoolean(xs)
...      // 与其他AnyVal类型相似的方法
implicit def refArrayOps[T <: AnyRef](xs: Array[T]): ArrayOps[T]
  = new ArrayOps.ofRef[T](xs)

@inline implicit def byteWrapper(x: Byte) = new runtime.RichByte(x)
...      // 与其他AnyVal类型相似的方法

implicit def genericWrapArray[T](xs: Array[T]): WrappedArray[T] = ...
implicit def wrapRefArray[T <: AnyRef](xs: Array[T]): WrappedArray[T] =...
implicit def wrapIntArray(xs: Array[Int]): WrappedArray[Int] = ...
...      // 与其他AnyVal类型相似的方法

implicit def wrapString(s: String): WrappedString = ...
implicit def unwrapString(ws: WrappedString): String = ...
```

如果你希望理解 runtime.Tuple2Zipped.Ops 方法的用意，首先你需要意识到大多数集合都提供了 zip 方法，该方法可以将两个集合连接起来。两个集合的元素组成一个新的元素的过程就好像合上拉链一样。

```
scala> val zipped = List(1,2,3) zip List(4,5,6)
zipped: List[(Int, Int)] = List((1,4), (2,5), (3,6))
```

我们可以对合并后的集合执行成对的操作，例如：

```
scala> val products = zipped map { case (x,y) => x * y }
products: List[Int] = List(4, 10, 18)
```

请注意我们在输入参数为元组类型的匿名函数中应用了模式匹配，传递给匿名函数的一对 Int 元素与 pair 元素相匹配。

Tuple2Zipper.Ops 和 Tuple3Zipper.Ops 提供了 invert 方法，该方法会将包含两个元素或三个元素的集合转换成包含二元元组或三元元组的集合。换言之，它们会压缩原本就定义在元组中的容器。例如：

```
scala> val pair = (List(1,2,3), List(4,5,6))
pair: (List[Int], List[Int]) = (List(1, 2, 3),List(4, 5, 6))

scala> val unpair = pair.invert
unpair: List[(Int, Int)] = List((1,4), (2,5), (3,6))

val pair = (List(1,2,3), List("one", "two", "three"))
tuple2ToZippedOps(pair) map {case (int, string) => (int*2, string.toUpperCase)}

val pair = (List(1,2,3), List(4,5,6))
pair map { case (int1, int2) => int1 + int2 }
```

在 Predef 对象中，还定义了许多转换成或转换自其他 Java 类型的转换方法，例如：

```
implicit def byte2Byte(x: Byte) = java.lang.Byte.valueOf(x)
implicit def Byte2byte(x: java.lang.Byte): Byte = x.byteValue
...      // Similar functions for the other AnyVal types.
```

为了能完整描述 Predef 中定义的隐式，我们再回顾一下之前在本章中见到的一些定义：

```
implicit def conforms[A]: A <:< A = ...
implicit def tpEquals[A]: A =:= A = ...
```

下面的代码将 java.util.Random 对象（http://docs.oracle.com/javase/8/docs/api/java/util/Random.html）转换成 scala.util.Random 对象（http://www.scala-lang.org/api/current/scala/util/Random.html）：

```
implicit def javaRandomToRandom(r: java.util.Random): Random = new Random(r)
```

scala.collection.convert 包中定义了一些 trait，这些 trait 提供了 Java 容器与 Scala 容器之间的转换方法。这为实现 Java 互操作性提供了便利。事实上，出于性能方面的考虑，Scala 会尽可能避免执行类型转换。不过因为目标集合的抽象体是建立在底层容器的基础上，因此并不会造成太多的性能损耗。

DecorateAsJava（http://www.scala-lang.org/api/current/scala/collection/convert/DecorateAsJava.html）中定义了一些 Scala 容器，这些容器类是 Java 容器的装饰类（decoration）。为了便于读者阅读，我们对代码中返回的类型标注进行了换行处理，而下面代码中出现的 ju 和 jl 则分别对应了 DecorateAsJava 实际源码中的 java.lang 及 java.util。各种名为 AsJava* 的类型则是提供了这些转换操作的辅助类型：

```
implicit def asJavaIteratorConverter[A](i : Iterator[A]):
  AsJava[ju.Iterator[A]] = ...
implicit def asJavaEnumerationConverter[A](i : Iterator[A]):
  AsJavaEnumeration[A] = ...
implicit def asJavaIterableConverter[A](i : Iterable[A]):
  AsJava[jl.Iterable[A]] = ...
implicit def asJavaCollectionConverter[A](i : Iterable[A]):
  AsJavaCollection[A] = ...
implicit def bufferAsJavaListConverter[A](b : mutable.Buffer[A]):
  AsJava[ju.List[A]] = ...
implicit def mutableSeqAsJavaListConverter[A](b : mutable.Seq[A]):
  AsJava[ju.List[A]] = ...
implicit def seqAsJavaListConverter[A](b : Seq[A]):
  AsJava[ju.List[A]] = ...
implicit def mutableSetAsJavaSetConverter[A](s : mutable.Set[A]):
  AsJava[ju.Set[A]] = ...
implicit def setAsJavaSetConverter[A](s : Set[A]):
  AsJava[ju.Set[A]] = ...
implicit def mutableMapAsJavaMapConverter[A, B](m : mutable.Map[A, B]):
  AsJava[ju.Map[A, B]] = ...
implicit def asJavaDictionaryConverter[A, B](m : mutable.Map[A, B]):
  AsJavaDictionary[A, B] = ...
implicit def mapAsJavaMapConverter[A, B](m : Map[A, B]):
  AsJava[ju.Map[A, B]] = ...
implicit def mapAsJavaConcurrentMapConverter[A, B](m: concurrent.Map[A, B]):
  AsJava[juc.ConcurrentMap[A, B]] = ...
```

我们可以使用 `DecorateAsScala`（http://www.scala-lang.org/api/current/scala/collection/convert/DecorateAsScala.html）中定义的隐式方法，将 Java 容器装饰成 Scala 容器：

```
implicit def asScalaIteratorConverter[A](i : ju.Iterator[A]):
  AsScala[Iterator[A]] = ...
implicit def enumerationAsScalaIteratorConverter[A](i : ju.Enumeration[A]):
  AsScala[Iterator[A]] = ...
implicit def iterableAsScalaIterableConverter[A](i : jl.Iterable[A]):
  AsScala[Iterable[A]] = ...
implicit def collectionAsScalaIterableConverter[A](i : ju.Collection[A]):
  AsScala[Iterable[A]] = ...
implicit def asScalaBufferConverter[A](l : ju.List[A]):
  AsScala[mutable.Buffer[A]] = ...
implicit def asScalaSetConverter[A](s : ju.Set[A]):
  AsScala[mutable.Set[A]] = ...
implicit def mapAsScalaMapConverter[A, B](m : ju.Map[A, B]):
  AsScala[mutable.Map[A, B]] = ...
implicit def mapAsScalaConcurrentMapConverter[A, B](m: juc.ConcurrentMap[A, B]):
  AsScala[concurrent.Map[A, B]] = ...
implicit def dictionaryAsScalaMapConverter[A, B](p: ju.Dictionary[A, B]):
  AsScala[mutable.Map[A, B]] = ...
implicit def propertiesAsScalaMapConverter(p: ju.Properties):
  AsScala[mutable.Map[String, String]] = ...
```

尽管这些方法都是定义在特征中的，不过 `JavaConverts` 对象（http://www.scala-lang.org/api/current/#scala.collection.JavaConverters$）为你提供导入这些方法的快捷方式，你可以使用下列语句把这些方法加载到当前作用域中：

```
import scala.collection.JavaConverters._
```

这些转换器的目的是让你能够对 Scala 容器调用 asJava 函数以生成对应的 Java 容器，对 Java 容器调用 asScala 以生成 Java 容器。因此，这些方法有效地定义了 Scala 容器类型与 Java 容器类型之间的 1 对 1 的关联关系。

但是更多时候，你希望能够选择输出容器的类型，而不是使用 asScala 或 asJava 方法提供的某些容器类型。这时，WrapAsJava 和 WrapAsScala 定义的一些额外的转换方法便能派得上用场。

下面列举了 WrapAsJava 中提供的一些方法：

```
implicit def asJavaIterator[A](it: Iterator[A]): ju.Iterator[A] = ...
implicit def asJavaEnumeration[A](it: Iterator[A]): ju.Enumeration[A] = ...
implicit def asJavaIterable[A](i: Iterable[A]): jl.Iterable[A] = ...
implicit def asJavaCollection[A](it: Iterable[A]): ju.Collection[A] = ...
implicit def bufferAsJavaList[A](b: mutable.Buffer[A]): ju.List[A] = ...
implicit def mutableSeqAsJavaList[A](seq: mutable.Seq[A]): ju.List[A] = ...
implicit def seqAsJavaList[A](seq: Seq[A]): ju.List[A] = ...
implicit def mutableSetAsJavaSet[A](s: mutable.Set[A]): ju.Set[A] = ...
implicit def setAsJavaSet[A](s: Set[A]): ju.Set[A] = ...
implicit def mutableMapAsJavaMap[A, B](m: mutable.Map[A, B]): ju.Map[A, B] =...
implicit def asJavaDictionary[A, B](m: mutable.Map[A, B]): ju.Dictionary[A, B]
implicit def mapAsJavaMap[A, B](m: Map[A, B]): ju.Map[A, B] = ...
implicit def mapAsJavaConcurrentMap[A, B](m: concurrent.Map[A, B]):
  juc.ConcurrentMap[A, B] = ...
```

下面列出了 WrapAsScala 中定义的方法：

```
implicit def asScalaIterator[A](it: ju.Iterator[A]): Iterator[A] = ...
implicit def enumerationAsScalaIterator[A](i: ju.Enumeration[A]):
  Iterator[A] = ...
implicit def iterableAsScalaIterable[A](i: jl.Iterable[A]): Iterable[A] = ...
implicit def collectionAsScalaIterable[A](i: ju.Collection[A]): Iterable[A]=...
implicit def asScalaBuffer[A](l: ju.List[A]): mutable.Buffer[A] = ...
implicit def asScalaSet[A](s: ju.Set[A]): mutable.Set[A] = ...
implicit def mapAsScalaMap[A, B](m: ju.Map[A, B]): mutable.Map[A, B] = ...
implicit def mapAsScalaConcurrentMap[A, B](m: juc.ConcurrentMap[A, B]):
  concurrent.Map[A, B] = ...
implicit def dictionaryAsScalaMap[A, B](p: ju.Dictionary[A, B]):
  mutable.Map[A, B] = ...
implicit def propertiesAsScalaMap(p: ju.Properties):
  mutable.Map[String, String] = ...
[source,scala]
```

与 JavaConverters 相似，我们可以使用 JavaConversions 对象（http://www.scala-lang.org/api/current/#scala.collection.JavaConversions$）将定义在这些 trait 中的方法导入到当前作用域：

```
import scala.collection.JavaConversions._
```

对容器执行排序是非常常见的操作，因此 Scala 提供了一些 Ordering[T] 类型的隐式值，其中 T 是一个 String 类型值，String 类型是一类 AnyVal 类型，可以转换成 Numeric 类

型或用户自定义的顺序值（数值类型是对作用于数值的常见操作的一种抽象）；请查看 Ordered[T]（http://www.scala-lang.org/api/current/scala/math/Ordering.html）中的定义。

我们不会列出一些在集合类型中定义的隐式 Ordering 值，不过在 Ordering[T] 类型中存在一些隐式定义：

```
implicit def ordered[A <% Comparable[A]]: Ordering[A] = ...
implicit def comparatorToOrdering[A](implicit c: Comparator[A]):Ordering[A]=...
implicit def seqDerivedOrdering[CC[X] <: scala.collection.Seq[X], T](
  implicit ord: Ordering[T]): Ordering[CC[T]] = ...
implicit def infixOrderingOps[T](x: T)(implicit ord: Ordering[T]):
  Ordering[T]#Ops = ...
implicit def Option[T](implicit ord: Ordering[T]): Ordering[Option[T]] = ...
implicit def Iterable[T](implicit ord: Ordering[T]): Ordering[Iterable[T]] =...
implicit def Tuple2[T1, T2](implicit ord1: Ordering[T1], ord2: Ordering[T2]):
   Ordering[(T1, T2)] = ...
...      // 相似的函数:从Tuple3到Tuple9
```

最后，还有一些转换可用于构建“迷你－DSL 语言”的并发或管理进程。

首先，scala.concurrent.duration 包（http://www.scala-lang.org/api/current/scala/concurrent/duration/package.html）中提供了一些用于定义持续时间的方法（我们会在第 17 章中看到这些类型以及用于调用并发的其他类型）：

```
implicit def pairIntToDuration(p: (Int, TimeUnit)): Duration = ...
implicit def pairLongToDuration(p: (Long, TimeUnit)): FiniteDuration = ...
implicit def durationToPair(d: Duration): (Long, TimeUnit) = ...
```

下面列举了一些各种各样的转换方法，代码摘自 scala.concurrent 包的多个文件：

```
// scala.concurrent.FutureTaskRunner:
implicit def futureAsFunction[S](x: Future[S]): () => S

// scala.concurrent.JavaConversions:
implicit def asExecutionContext(exec: ExecutorService):
  ExecutionContextExecutorService = ...
implicit def asExecutionContext(exec: Executor): ExecutionContextExecutor = ...

// scala.concurrent.Promise:
private implicit def internalExecutor: ExecutionContext = ...

// scala.concurrent.TaskRunner:
implicit def functionAsTask[S](fun: () => S): Task[S] = ...

// scala.concurrent.ThreadPoolRunner:
implicit def functionAsTask[S](fun: () => S): Task[S] = ...
implicit def futureAsFunction[S](x: Future[S]): () => S = ...
```

最后，Process 提供了一些与 UNIX shell 命令类似的方法，以支持操作系统进程操作：

```
implicit def buildersToProcess[T](builders: Seq[T])(
  implicit convert: T => Source): Seq[Source] = ...
implicit def builderToProcess(builder: JProcessBuilder): ProcessBuilder = ...
implicit def fileToProcess(file: File): FileBuilder = ...
implicit def urlToProcess(url: URL): URLBuilder = ...
```

```
implicit def stringToProcess(command: String): ProcessBuilder = ...
implicit def stringSeqToProcess(command: Seq[String]): ProcessBuilder = ...
```

这真是一个很长的代码！不过我希望你可以通过浏览这些代码对 Scala 库是如何支持并运用隐式能有一个大概的了解。

## 5.8　合理使用隐式

在构建 DSL 以及简化代码的 API 的过程中，隐式参数机制是一种非常强大的工具。不过由于传递的隐式参数以及隐式值几乎是不可见的，这就增加了理解代码的难度。所以，我们应该合理地使用隐式。

有一种可以提高隐式可见性的方法，即将隐式值统一放到名为 `implicts` 的特殊包或名为 `Implicits` 的对象中。这种方式使得读者一旦遇到导入语句中的 `implicit` 字样时，便会留意到除了 Scala 内置的隐式之外，还存在一些其他的隐式。值得庆幸的是，目前有些 IDE 也能够指出代码中存在的隐式。

## 5.9　本章回顾与下一章提要

我们在本章中深入学习了 Scala 语言中关于隐式的各种知识。我希望你能在理解了隐式的功能和用途的同时，还能牢记它们的缺点。

接下来，我们将深入学习函数式编程的原理。我们首先会讨论函数式编程的核心概念并解释它们的重要性。之后我们将会了解到 Scala 库中大多数容器类型所提供的强大的函数。通过学习，我们能够掌握如何使用这些函数构建简洁却又强大的程序。

# 第6章

# Scala函数式编程

用 100 个函数操作 1 个数据结构，总比用 10 个函数操作 10 个数据结构来得好。

——Alan J. Perlis

每隔一二十年，就会有一种计算机思想成为主流。而在成为主流之前，这种思想可能已经在计算机科学研究领域或者工程实践的某个幽暗角落里潜伏了好几十年之久。之所以能成为主流思想是由于它可以恰当地解决当前面临的问题。面向对象编程语言（OOP）发明于 20 世纪 60 年代，由于面向对象编程很适合解决图形界面的设计问题，它因此成为 20 世纪 80 年代的主流编程范式。

函数式编程走向主流的过程也非常类似。关于函数式编程的研究实际上比面向对象编程的还要久远。函数式编程主要可以为当前面临的三大挑战提供解决方案。

(1) 是并发的普遍需求。有了并发，我们可以对应用进行水平扩展，并提供其对抗服务器故障的能力。所以，如今并发编程已经是每个开发者的必备技能了。

(2) 是编写数据导向（如“大数据”）程序的要求。当然，从某种意义上说，每个程序都与数据密切相关，但如今大数据的发展趋势，使得有效处理海量数据的技术被提高到了更重要的位置。

(3) 是编写无 bug 的程序的要求。这个挑战与编程本身一样古老，但函数式编程从数学的角度为我们提供了新的工具，使我们向无 bug 的程序又迈进了一步。

状态不可变这一特点解决了并发编程中最大的难题，即对共享的可变状态的访问问题。因此，编写状态不可变的代码就称为编写健壮的并发程序的必备品，而拥抱函数式编程就是写出状态不可变代码的最好途径。状态不可变，以及严密的函数式编程思想有其数学理论为基础，还能减少程序中的逻辑错误。

在本章和以后的章节中，我们会逐渐掌握函数式编程的操作方法，同时我们也会发现函数

式编程在处理数据导向的应用中的明显优势。这一点会在第 18 章深入讨论。本书中讨论的很多主题都有助于减少 bug 的产生，特别有帮助的部分我们会进行特别讲解。

到目前为止，在本书中我假定读者对面向对象编程有基本的概念。但由于理解函数式编程的人相对较少，所以我们会花些时间介绍函数式编程的基本概念。在 17.3 节，我们会深入探讨函数式编程是编写并发程序的有效工具这一问题，另外，你还将发现，它也对改进面向对象程序也很有帮助。

在本章的开始，我们会暂时抛开 Scala 语言，花少量篇幅介绍函数式编程。对于 Scala 这样一门混合范式的语言，函数式编程并不是必须采用的，但凡是能用得上的地方，都推荐你采用 Scala。

# 6.1　什么是函数式编程

存不存在一门编程语言不使用函数或其他类似函数的结构？不管它们叫方法（method），程序（procedure），还是 `GOTO`，它们都相当于函数。所有的语言都有函数。

函数式编程的理论基础是数学中关于函数和值的规则。这对软件编程中的函数有着深远的影响。

## 6.1.1　数学中的函数

在数学里，函数没有副作用。以下是经典的 sin(*x*) 函数：

$y=\sin(x)$

无论 sin(*x*) 做了多少计算，它的所有结果都被函数返回并赋值给了 *y*。在 sin(*x*) 内部，没有任何全局状态被修改。这样，我就称该函数是无副作用函数，即纯函数。

纯函数极大地简化了函数的分析、测试和调试。你可以不考虑调用该函数的上下文信息，否则的话，就要受该上下文中调用的其他函数的影响了。

由于可以忽略上下文信息，引用是透明的，这带来了两个结果。第一，你可以在任何地方调用函数，并确信其行为与上下文无关，每次的行为都能够确保相同。由于没有任何全局对象被修改，对函数的并发调用也是安全可靠的，不需要任何线程安全的编写技巧。

第二，你可以用表达式所计算得出的值替换表达式本身。考虑如下例子：由于方程 `sin(pi/2) = 1` 成立，只要 sin 函数是纯函数，代码分析器如编译器或者运行时的虚拟机就可以将函数调用 `sin(pi/2)` 替换为 1，而不会引入任何错误。

> 返回 `Unit` 的函数只能执行带副作用的操作，它必须在某处对可变状态做修改。一个例子就是调用了 `println` 或 `printf` 去打印 the world 字符串的函数，它将某个 I/O 相关状态修改为 the world。

由于我们可以用其中一个替换另一个，值与函数之间有着天然的对称关系。那么能否用函数来替换值，或者将函数视为值呢？

事实上，在函数式编程中，函数是第一等级的值，就像数据变量的值一样。你可以从函数中组合形成新函数（如 $\tan(x)=\sin(x)/\cos(x)$），可以将函数赋值给变量，也可以将函数作为参数传递给其他函数，还可以将函数作为其他函数的返回值。

当一个函数采用其他函数作为变量或返回值时，它被称为*高阶函数*。在数学中，微积分中有两个高阶函数的例子——微分与积分。我们将一个表达式作为函数传给“微分函数”，然后微分函数返回了一个新函数，即原函数的导数。

我们已经接触了不少高阶函数的例子，如：集合类型的 `map` 方法。`map` 方法的参数是一个函数，该函数对集合中的每个元素进行操作。

## 6.1.2 不可变变量

“变量”一词在函数式编程中有新的含义。传统的面向对象编程是面向过程编程的一个子集，如果你有面向过程编程的背景，你会认为变量就是可变的。然而，在函数式编程中，变量是不可变的。

这是数学原理带来的另一个结果。在表达式 $y=\sin(x)$ 中，一旦 $x$ 确定，$y$ 也就确定了。类此地，值也是不可变的。如果你想对 3 加 1，你不能“对 3 这个对象做修改”，而只能“创建一个新的值表示 4”。我们在这里用“值”一词作为不可变变量的同义词。

假如一开始你对不可变变量不太习惯，对于值的不可变特性你也会很难适应。如果你没法修改变量，你就不能用循环变量了，也不能通过方法的调用修改对象的状态，不能执行输入输出操作。用值不可变的特性思考需要花些力气。

显然，我们不可能永远使用纯函数。没有了输入输出，你的计算机除了发热以外别无用处。实践中的函数式编程对何时执行可变操作，什么时候只调用纯函数做了明确界定。

### 为什么输入输出是有副作用的？

很容易想到 `sin(x)` 是没有副作用的纯函数。为什么输入和输出是有副作用的非纯操作？它们修改了我们周围的状态，如：文件的内容和屏幕上看到的输出。它们也不是引用透明的。每次调用 `readline`（定义于 `Predef` 中），都会取出不同的结果。每次调用 `println`（同样定义于 `Predef`），会传入不同的值，但都返回 `Unit`。

这并不意味着函数式编程完全没有状态。如果那样的话，函数式编程也就没什么用处了。你可以用新的对象或新开的栈空间来表示状态的修改，即调用函数，并返回你要的值。

回顾第 2 章的例子：

```
// src/main/scala/progscala2/typelessdomore/factorial.sc

def factorial(i: Int): Long = {
  def fact(i: Int, accumulator: Int): Long = {
    if (i <= 1) accumulator
    else fact(i - 1, i * accumulator)
  }
```

```
    fact(i, 1)
  }

  (0 to 5) foreach ( i => println(factorial(i)) )
```

我们用递归来计算阶乘。对计算结果的每次更新被压到了栈上，而不是直接修改栈中的值。

在这个例子的最后，我们将结果打印出来。所有的函数式语言都提供了类似的机制，可以帮助我们完成 I/O 操作，以及其他必须涉及状态修改的行为。对于 Scala 这样非常灵活的混合范式语言，我们需要学会审慎规范地在必须修改状态时才对状态做修改，剩下的部分应该尽量做到无副作用。

值不可变性对编写可扩展的并发程序有巨大的好处。多线程程序的大部分难点在于对公共的可变状态进行访问时的同步问题。如果去掉了公共状态的可变性，这个问题将不复存在。引用透明的函数和值不可变性的结合，使得函数式编程成为编写并发软件应用的更好选择。对并发程序进行扩展的需求的增长，提高了业界对函数式编程的关注。

这两个特点对编程还有其他方面的好处。60 年间，我们发明的所有编程语言的构造都在试图管理复杂性。高阶的纯函数被称为粘合剂，它们可以将灵活、细粒度的代码块和粒度更大、更复杂的程序很好地组合在一起。我们已经遇见过与集合有关的方法串在一起并用少量代码完成复杂逻辑的例子。

纯函数与值不可变性极大地降低了 bug 出现的概率。致命的 bug 大多来源于可变状态，尤其是那种部署到生产环境之前很难测试出来的 bug，这种 bug 通常也是最难修复的。

可变状态意味着一个模块对状态的修改，对于其他模块来说是不可预见的，由此可能引发“幽灵般的超距作用”。

纯函数通过去掉面向对象的程序代码中的同步保护代码，从而大大简化了代码。对于面向对象编程而言，对数据结构的访问往往被封装在对象中。因为如果状态可变，我们不可能简单地将其与客户端共享，而往往通过增加特定的访问方法，从而使得客户端对状态的访问在我们的控制范围之内。这些用于访问的方法增加了代码的体积，从而增加了测试与维护的成本。这也使得 API 增多，增加了客户端的学习成本。

当我们有了值的不可变性以后，这些问题基本都消失了。我们可以将内部数据设为可对外公开访问，从而不必再担心数据的值被修改的问题。当然了，减少耦合和暴露抽象的普遍原则依然适用。隐藏实现细节对于最小化 API 签名依然是非常重要的。

一个关于不可变性的反常现象是，其程序反而比状态可变的程序更快。如果你不能修改对象的值，你就只能在需要改变值的时候复制一份，再进行修改，对吗？幸运的是，函数式编程通过共享对象中的未修改部分使得复制的开销最小化。

与此相反，面向对象语言中的数据结构并不支持高效的复制操作。当需要共享数据结构的内部状态时，为了使数据免受不需要的修改操作所污染，客户端不得不对可变的数据结构做代价昂贵的复制。

另一个性能提供的原因在于数据结构的惰性求值，如 Scala 的 `Stream` 类型（http://www.

scala-lang.org/api/current/scala/collection/immutable/Stream.html）。少部分函数式语言如 Haskell，它们的默认为惰性值，这意味着求值操作被推迟到对应的值的确需要用的时候[1]。

Scala 默认的求值规则为立即求值或者严格求值。惰性求值在 Scala 中的优势在于它可以避免做不需要的求值计算。例如：处理很大的流式数据时，如果只需要其中开头的一小部分数据，那么对整个数据做处理就显得十分浪费。即使最终的确需要整块数据，惰性策略也可以提前得出结果，而不需要等待整块数据都处理完成才行。

那么，Scala 为何不干脆也将惰性求值设为默认的规则呢？这是由于在很多场景下惰性求值的效率并不高，而且求值的结果也很难预测。所以，多数函数式编程语言默认为立即求值。但场景合适时也可以使用惰性求值，如在处理流式数据时。

是时候来探讨 Scala 中的函数式编程实践了。我们会继续讨论函数式的其他方面特点及其好处。在介绍 Scala 的面向对象特性之前，我们先介绍其函数式特性，以鼓励大家真正理解函数式的好处。对于 Java 程序员而言，最便捷的学习途径就是先将 Scala 视为“更好的 Java”，一门拥有一些奇怪隐晦的函数式特点的面向对象编程的语言。我会将这些隐晦的角落清楚地展现在大家面前，从而使得我们得以领略其魅力和实力。

本章介绍了那些我认为每个 Scala 程序员都必须掌握的基础。函数式编程是一个庞大而丰富的领域，我们将会在第 16 章为新手介绍一些更高级但并非必须掌握的话题。

## 6.2 Scala中的函数式编程

作为一门面向对象与函数式的混合范式语言，Scala 并不强制要求函数必须是纯函数，也不要求变量不可变。尽管它的确推荐你在任何可能的情况下这么做。

我们来快速概述一下我们已经学到的东西。

以下是几个高阶函数，我们将其组合在一起，用它来对一个整数列表进行遍历，过滤出其中的偶数，对每个偶数乘以 2，再使用 `reduce` 函数将各个整数乘在一起：

```
// src/main/scala/progscala2/fp/basics/hofs-example.sc

(1 to 10) filter (_ % 2 == 0) map (_ * 2) reduce (_ * _)
```

结果为 122880。

回顾一下，`_ % 2 == 0`，`_ * 2`，与 `_ * _` 是函数字面量。前两个函数只有一个参数，赋值给占位符 `_`；最后一个函数带两个参数，该函数本身是 `reduce` 函数的参数。

`reduce` 函数将各个元素做累乘，也就是说它将整数的集合 reduce 为一个值。`reduce` 函数带两个参数，均赋值给了占位符 `_`。其中一个参数是集合中的当前元素，另一个参数就是累乘值，是上一次调用 `reduce` 函数得到的部分元素的累乘结果。（第一个参数是累乘参数，还是第二个参数是累乘参数取决于具体实现）对传入的函数的要求是：其计算必须满足结合律，类似乘法与加法，因为我们不保证集合中元素的计算顺序。

注 1：Haskell 社区有一个笑话，说他们总是将成功尽可能地推迟到最后一分钟。

于是，我们只用了一行代码，没有用可变的计数器，也没有用可变变量作为累乘结果，就“循环”遍历了列表得出结果。

### 6.2.1 匿名函数、Lambda与闭包

对刚才的例子做以下修改：

```
// src/main/scala/progscala2/fp/basics/hofs-closure-example.sc

var factor = 2
val multiplier = (i: Int) => i * factor

(1 to 10) filter (_ % 2 == 0) map multiplier reduce (_ * _)

factor = 3
(1 to 10) filter (_ % 2 == 0) map multiplier reduce (_ * _)
```

我们定义一个名为 `factor` 的变量，作为累乘因子。而之前的匿名函数 `_ * 2` 则替换为一个名为 `multiplier` 的变量，变量的值由 `factor` 决定。注意，`multiplier` 事实上也是一个函数。由于函数在 Scala 中是第一等的，因此我们定义了表示函数的变量。不过，这不是简单的替换，在这里 `multiplier` 引用了 `factor`，而不是将其硬编码为 2。

注意看我们使用两个不同的 `factor` 值时，程序的运行结果。首先我们的输出值为 122880，与之前相同，但接着输出值为 933120。

尽管 `multiplier` 是一个不可变的函数字面量，当 `factor` 改变时，`multiplier` 的行为也跟着改变。

在 `multiplier` 函数中有两个变量 `i` 和 `factor`。`i` 是一个函数的参数，所以每次调用时，`i` 都绑定了一个新的值。

然而，`factor` 并不是 `multiplier` 的参数，而是一个自由变量，是一个当前作用域中某个值的引用。所以，编译器创建了一个闭包，用于包含（或“覆盖”）`multiplier` 与它引用的外部变量的上下文信息，从而也就绑定了外部变量本身。

这就是 `factor` 变化时，`multiplier` 也跟着变化的原因。`Multiplier` 引用了 `factor`，每次调用时都重新读取 `factor` 的值。如果函数没有外部引用，那它就只是包含了自身，不需要外部上下文信息。

即使 `factor` 处于某个局部作用域（如某个方法）中，而我们将 `multiplier` 传递给其他作用域（如另一个方法）中时，这一机制仍然有效。该自由变量 `factor` 的有效性一直伴随 `multiplier` 函数：

```
// src/main/scala/progscala2/fp/basics/hofs-closure2-example.sc

def m1 (multiplier: Int => Int) = {
  (1 to 10) filter (_ % 2 == 0) map multiplier reduce (_ * _)
}

def m2: Int => Int = {
  val factor = 2
```

```
  val multiplier = (i: Int) => i * factor
  multiplier
}

m1(m2)
```

我们调用 m2，返回了一个类型为 Int => Int 的函数。返回的内部值是 multiplier，multiplier 引用了 m2 中定义的变量 factor，一旦 m2 返回，就离开了 factor 变量的作用域。

然后调用 m1，将 m2 的返回值传递给它。尽管 factor 变量已经离开了 m1 的作用域，但程序的输出与之前的例子相同，仍为 122880。m2 返回的函数事实上是一个闭包，它包含了对 factor 的引用。

这里有几个概念存在交叉的常用术语。

- 函数

  一种具有名或匿名的操作。其代码直到被调用时才执行。在函数的定义中，可能有也可能没有引用外部的未绑定变量。

- Lambda

  一种匿名函数。在它的定义中，可能有也可能没有引用外部的未绑定变量。

- 闭包

  是一个函数，可能匿名或具有名称，在定义中包含了自由变量，函数中包含了环境信息，以绑定其引用的自由变量。

### 为什么用 Lambda 这个名字？

用 Lambda 来表示匿名函数源于 Lambda 微积分，Lambda 微积分中用希腊字母 Lambda（λ）表示匿名函数。Alonzo Church 在数学的可计算性理论中首先研究了 Lambda 微积分，在 Lambda 微积分中，将函数的特性总结为：函数是绑定了值，或用其他表达式替换变量时可被求值或可被应用（apply）的计算行为的抽象。（apply 一词就是我们已经接触过的默认方法 apply 名字的来源。）Lambda 微积分也为函数表达式的简化、如何用值代替变量等定义了规则。

不同的编程语言往往用上述的术语和其他术语来表示这几个有细微区别的概念。在 Scala 中，我们称 Lambda 函数为匿名函数或函数字面量，对于闭包和其他函数则并不区分（除非这样的区别的确重要）。

**作为函数的方法**

在上一小节讨论函数捕捉外部变量时，我们将匿名函数 multiplier 定义为一个值：

```
val multiplier = (i: Int) => i * factor
```

不过，你也可以用方法代替函数：

```
// src/main/scala/progscala2/fp/basics/hofs-closure3-example.sc

object Multiplier {
```

```
  var factor = 2
  // Compare: val multiplier = (i: Int) => i * factor
  def multiplier(i: Int) = i * factor
}

(1 to 10) filter (_ % 2 == 0) map Multiplier.multiplier reduce (_ * _)

Multiplier.factor = 3
(1 to 10) filter (_ % 2 == 0) map Multiplier.multiplier reduce (_ * _)
```

`multiplier` 此时是一个方法。比较一下函数定义与方法定义的语法区别。除了 `multiplier` 是方法以外，我们对它的使用与函数相同，因此它并没有引用 `this`。在需要函数的地方用了方法，我们就称该方法被提升为了函数。“提升”这个词的其他用法我们会在后续章节中继续学习。

### 6.2.2 内部与外部的纯粹性

如果我们用相同的 $x$ 值调用 $\sin(x)$ 函数一千次，系统每次都重新进行一次将会是极大的浪费。即使在“纯粹”的函数库中，也常常执行内部优化，如缓存（或称为“记住”）之前的计算结果。但缓存引入了副作用，因此缓存的状态会被修改。

然而，这种状态的改变对用户来说是不可见的（除非影响性能的意义）。函数的实现只需要负责达到“契约”即可，即线程安全与透明引用。

## 6.3 递归

在函数式编程中，递归比在命令式编程中更为重要。递归是实现“循环”的唯一方法，因为你无法修改循环变量。

阶乘的计算就是一个很好的例子。以下是 Java 用命令式循环的一种阶乘实现：

```
// src/main/java/progscala2/fp/loops/Factorial.java
package progscala2.fp.loops;

public class Factorial {
  public static long factorial(long l) {
    long result = 1L;
    for (long j = 2L; j <= l; j++) {
      result *= j;
    }
    return result;
  }

  public static void main(String args[]) {
    for (long l = 1L; l <= 10; l++)
      System.out.printf("%d:\t%d\n", l, factorial(l));
  }
}
```

循环变量 `j` 和计算结果均为可变变量。（为了简化程序，我们忽略了输入变量小于等于零的情况。）该程序使用 `sbt` 构建，因此我们可以在 `sbt` 提示符下运行它：

```
> run-main progscala2.fp.loops.Factorial
[info] Running FP.loops.Factorial
1:  1
2:  2
3:  6
4:  24
5:  120
6:  720
7:  5040
8:  40320
9:  362880
10:  3628800
[success] Total time: 0 s, completed Feb 12, 2014 6:12:18 PM
```

以下为递归实现：

```
// src/main/scala/progscala2/fp/recursion/factorial-recur1.sc
import scala.annotation.tailrec

// What happens if you uncomment the annotation??
// @tailrec
def factorial(i: BigInt): BigInt =
  if (i == 1) i
  else i * factorial(i - 1)
}

for (i <- 1 to 10)
  println(s"$i:\t${factorial(i)}")
```

输出是相同的，但这里没有可变的变量。（你可能认为在最后测试部分的 for 表达式中 i 是可变的，但实际上它不是，我们将会在第 7 章进行阐释。）

递归是表达函数的最常用方式。然而，递归也有两个缺点：反复调用函数带来的开销；栈溢出的风险。

如果编译器或运行环境能够将我们写出的纯粹的且以递归实现的函数优化为循环就好了。接下来我们就将探讨这个话题。

## 6.4 尾部调用和尾部调用优化

有一种特殊的递归被称为尾递归。在尾递归中，函数可以调用自身，并且该调用是函数的最后一个（“尾部”）操作。尾递归非常重要，因为这是能把函数优化为循环的最重要的一种递归。循环可以消除潜在的栈溢出风险，同时也因为消除了函数调用开销而提升效率。尽管目前原生 JVM 还不支持尾递归优化，但 scalac 可以。

然而，我们计算阶乘的例子并不是尾递归。阶乘调用了自身以后，还将结果相乘。回忆一下，第 2 章中讲到 Scala 有一个标记 @tailrec（http://www.scala-lang.org/api/current/#scala.annotation.tailrec），可以添加在你认为是尾递归的递归函数中。如果编译器无法对它做尾递归优化，系统将抛出异常。

将之前的例子中 @tailrec 前的 // 注释符号去掉，运行并观察输出。

幸运的是，我们在嵌套方法定义与递归中的例子是一个合法的尾递归实现。以下是其改进版：

```
// src/main/scala/progscala2/fp/recursion/factorial-recur2.sc
import scala.annotation.tailrec

def factorial(i: BigInt): BigInt = {
  @tailrec
  def fact(i: BigInt, accumulator: BigInt): BigInt =
    if (i == 1) accumulator
    else fact(i - 1, i * accumulator)

  fact(i, 1)
}

for (i <- 1 to 10)
  println(s"$i:\t${factorial(i)}")
```

以上脚本的输出与之前的例子相同。这样，`factorial` 用嵌套的方法 `fact` 完成了计算工作。`fact` 是尾递归的，因为它用一个累乘的参数去保存计算的结果。该参数与一个乘数相乘，然后在函数的尾部调用 `fact` 自身。`@tailrect` 标记不再抛出异常，因为编译器可以成功将其转为循环。

如果你用一个大数——如 10 000——去调用原始的非尾递归版本的 `factorial` 函数，一般普通的台式计算机会出现栈溢出。尾递归优化的版本则可以成功运行，并返回结果。

定义一个嵌套的尾递归函数，将累积值作为参数，是将很多普通递归算法转为尾递归的实用技巧。

> 当一个调用了自身的方法，有可能被子类型中的同名方法覆写时，尾递归是无效的。所以，尾递归的方法必须用 `private` 或 `final` 关键字声明，或者将它嵌套在另一个方法中。

## 尾递归的trampoline优化

trampoline（原意为“蹦床”）是指通过依次调用各个函数完成一系列函数之间的循环。暗喻在多个函数之间反复来回调用。

我们考虑一下这种递归：函数 `A` 不调用自身，而是调用了另一个函数 `B`；而函数 `B` 又调用了 `A`，`A` 再次调用 `B`，如此循环。trampoline 可以将这种反复来回调用的函数也转化为循环。由此可见，trampoline 是一种无需递归就可以处理这种来回调用的计算的数据结构。

Scala 库中有一个尾递归对象（http://www.scala-lang.org/api/current/scala/util/control/TailCalls$.html）用于达到这个目的。

以下代码提供了确定一个数是否为偶数的方法，但效率不高。（因为 `isEven` 和 `isOdd` 互相引用。可以用 REPL 的 `:paste` 模式粘贴以下代码。）

```
// src/main/scala/progscala2/fp/recursion/trampoline.sc
// From: scala-lang.org/api/current/index.html#scala.util.control.TailCalls$
```

```
import scala.util.control.TailCalls._

def isEven(xs: List[Int]): TailRec[Boolean] =
  if (xs.isEmpty) done(true) else tailcall(isOdd(xs.tail))

def isOdd(xs: List[Int]): TailRec[Boolean] =
 if (xs.isEmpty) done(false) else tailcall(isEven(xs.tail))

for (i <- 1 to 5) {
  val even = isEven((1 to i).toList).result
  println(s"$i is even? $even")
}
```

以上代码对列表中的元素进行来回调用，如果到列表结束时，处于 isEven 方法中，则返回 true；如果在 isOdd 方法中，则返回 false。

运行脚本，得到以下与期望相符的输出结果：

```
1 is even? false
2 is even? true
3 is even? false
4 is even? true
5 is even? false
```

## 6.5 偏应用函数与偏函数

考虑如下带两个参数列表的简单方法：

```
// src/main/scala/progscala2/fp/datastructs/curried-func.sc

scala> def cat1(s1: String)(s2: String) = s1 + s2
cat1: (s1: String)(s2: String)String
```

如果我们需要一个专门的版本，要求第一个字符串总是 Hello，我们可以通过偏应用函数来定义这样的函数：

```
scala> val hello = cat1("Hello ") _
hello: String => String = <function1>

scala> hello("World!")
res0: String = Hello World!

scala> cat1("Hello ")("World!")
res1: String = Hello World!
```

REPL 的输出表明，hello 是一个 <function1>，也就是带一个参数的函数。

我们调用 cat1 时给出了第一个参数列表，后面跟上一个下划线（_），用它来定义了 hello。下面我们试着不用下划线调用 cat1：

```
scala> val hello = cat1("Hello ")
>console<:8: error: missing arguments for method cat1;
follow this method with `_' if you want to treat it as a
partially applied function
```

```
val hello = cat1("Hello ")
                ^
```

关键就在于偏应用函数。对于拥有多个参数列表的函数而言，如果你希望忽略其中一个或多个参数列表，可以通过定义一个新函数来实现。也就是说，你给出了部分所需的参数。为了避免潜在的表达式歧义，Scala 要求在后面加上下划线，用来告诉编译器你的真实目的。注意，这个特性只对函数的多个参数列表有效，对一个参数列表中的多个参数的情况并不适用。

编写 `cat1("Hello")("World")` 时，它的开头有点像偏作用函数 `cat1("Hello")_`，但随后我们给出了第二个参数列表，这样就消除了歧义。

下面厘清一些让人困惑的术语。这里我们一直在讨论偏应用函数，其实偏应用函数表示表达式中使用了函数，但并未给出所需的所有参数列表。所以，系统返回了一个新的函数，该函数的参数列表是原函数中没有给出的剩下的那部分参数列表。

我们也掌握了偏函数的概念，这点在 2.4 节有所介绍。回顾一下，偏函数带一个某类型参数，但函数并未针对该类型的所有值实现相应处理逻辑。考虑下例：

```
scala> val inverse: PartialFunction[Double,Double] = {
     |   case d if d != 0.0 => 1.0 / d
     | }
inverse: PartialFunction[Double,Double] = <function1>

scala> inverse(1.0)
res128: Double = 1.0

scala> inverse(2.0)
res129: Double = 0.5

scala> inverse(0.0)
scala.MatchError: 0.0 (of class java.lang.Double)
  at scala.PartialFunction$$anon$1.apply(PartialFunction.scala:248)
  at scala.PartialFunction$$anon$1.apply(PartialFunction.scala:246)
  ...
```

这个求倒数的函数并不太健壮，因为当 `d` 非常小但非零时，`1/d` 也会崩溃！当然，这里想说的是，这个 `inverse` 函数是“偏”的，只处理除以 `0.0` 外的所有 `Double` 类型。

> 偏作用函数是一个表达式，带部分而非全部参数列表的函数。返回值是一个新函数，新函数负责携带剩下的参数列表。偏函数则是单参数的函数，并未对该类型的所有值都有定义。偏函数的字面量语法由包围在花括号中的一个或多个 `case` 语句构成。

## 6.6 Curry化与函数的其他转换

在 2.5.2 节，我们引入了“方法可以拥有多个参数列表”的思想。我们讨论过它的用法，不过函数还有另一种基本性质也支持这一思想，称为 Curry。该命名来自于数学家 Haskell

Curry（这也是 Haskell 的命名来源）。事实上，Curry 的研究来源于 Moses Schönfinkel 的思想，不过，是 Schönfinkeling 还是 Schönfinkelization，就更难解释了……

Curry 将一个带有多参数的函数转换为一系列函数，每个函数都只有一个参数。

在 Scala 中，Curry 化的函数带有多个参数列表，处理后，每个函数都只有一个参数列表。回顾一下上一节定义的 cat1 方法：

```
// src/main/scala/progscala2/fp/datastructs/curried-func.sc

def cat1(s1: String)(s2: String) = s1 + s2
```

也可以用以下语法来定义 curry 化的函数：

```
def cat2(s1: String) = (s2: String) => s1 + s2
```

第一种语法更具可读性。而第二种语法在将 Curry 化的函数作为偏应用函数时，不需要在后面加上下划线：

```
scala> def cat2(s1: String) = (s2: String) => s1 + s2
cat2: (s1: String)String => String

scala> val cat2hello = cat2("Hello ") // 没有 _
cat2hello: String => String = <function1>

scala> cat2hello("World!")
res0: String = Hello World!
```

调用两个函数的写法相同，返回结果也相同：

```
scala> cat1("foo")("bar")
res0: String = foobar

scala> cat2("foo")("bar")
res1: String = foobar
```

我们还可以将一个带有多个参数的方法转为 Curry 化的形式（注意如何用偏应用函数的语法来完成 Curry 化）：

```
scala> def cat3(s1: String, s2: String) = s1 + s2
cat3: (s1: String, s2: String)String

scala> cat3("hello", "world")
res2: String = helloworld

scala> val cat3Curried = (cat3 _).curried
cat3Curried: String => (String => String) = <function1>

scala> cat3Curried("hello")("world")
res3: String = helloworld
```

在这个例子中，我们将带有两个参数的方法 cat3，转为其 Curry 化的等价函数，该函数带两个参数列表。如果 cat3 带三个参数，则相应的 Curry 也就带有三个参数列表，以此类推。

注意类型签名 cat3Curried，String => (String => String)。我们这次用函数值来代替：

```
// src/main/scala/progscala2/fp/datastructs/curried-func2.sc

scala> val f1: String =>  String => String  =
    (s1: String) => (s2: String) => s1+s2
f1: String => (String => String) = <function1>

scala> val f2: String => (String => String) =
    (s1: String) => (s2: String) => s1 + s2
f2: String => (String => String) = <function1>

scala> f1("hello")("world")
res4: String = helloworld

scala> f2("hello")("world")
res5: String = helloworld
```

类型签名 `String => String => String` 与 `String => (String => String)` 是等价的。调用 `f1` 或 `f2` 时绑定第一个参数列表，将会返回一个类型为 `String => String` 的新函数。

我们也可以用 `Function`（http://www.scala-lang.org/api/current/scala/Function$.html）中的一个方法对函数做“去 Curry”：

```
scala> val cat3Uncurried = Function.uncurried(cat3Curried)
cat3Uncurried: (String, String) => String = <function2>

scala> cat3Uncurried("hello", "world")
res6: String = helloworld

scala> val ff1 = Function.uncurried(f1)
ff1: (String, String) => String = <function2>

scala> ff1("hello", "world")
res7: String = helloworld
```

Curry 的一个实际用处是对特定类型的数据函数做特殊化。函数可以接受通用的类型，而 Curry 化的函数形式则只接受特定的类型。

以下就是这种方法的一个示例。原函数用于计算连乘，Curry 化的函数是原函数的特化版本：

```
scala> def multiplier(i: Int)(factor: Int) = i * factor
multiplier: (i: Int)(factor: Int)Int

scala> val byFive = multiplier(5) _
byFive: Int => Int = <function1>

scala> val byTen = multiplier(10) _
byTen: Int => Int = <function1>

scala> byFive(2)
res8: Int = 10

scala> byTen(2)
res9: Int = 20
```

原函数是 multiplier，带两个参数：第一个为整数，第二个也是整数，表示系数。我们随后将其 Curry 化为两个函数变量值。不要忘了后面的下划线。接着，我们就调用了这两个函数。

正如你所看到的那样，Curry 与偏应用函数是紧密相关的两个概念。你可能会发现这两个概念可以互相替换，但重要的是它们的应用。

此外，还有另外一些值得了解的函数转换形式。

你可能会遇到这样一种场景：你有一个元组，例如，一个三元素元组，而你需要调用一个包含三个参数的函数：

```
scala> def mult(d1: Double, d2: Double, d3: Double) = d1 * d2 * d3
mult: (d1: Double, d2: Double, d3: Double)Double

scala> val d3 = (2.2, 3.3, 4.4)
d3: (Double, Double, Double) = (2.2,3.3,4.4)

scala> mult(d3._1, d3._2, d3._3)
res10: Double = 31.944000000000003
```

代码不太美观，但是因为需要使用元组字面量语法，例如：(2.2, 3.3, 4.4)，进行转换，元组元素与函数的参数列表变得很相称。我们需要一个新版的 mult 函数，其参数就是一个三元素的元组。幸运的是，Function 对象为我们提供了元组形式和非元组形式的方法：

```
scala> val multTupled = Function.tupled(mult _)
multTupled: ((Double, Double, Double)) => Double = <function1>

scala> multTupled(d3)
res11: Double = 31.944000000000003

scala> val multUntupled = Function.untupled(multTupled)
multUntupled: (Double, Double, Double) => Double = <function3>

scala> multUntupled(d3._1, d3._2, d3._3)
res12: Double = 31.944000000000003
```

注意当我们将 mult 传递给 Function.tupled 时，是如何做偏应用化的。但是，假如我们将得到的函数值传递给 Function 对象的其他方法时，就不再需要这种语法了。这种语法现象是面向对象的方法与函数式编程的函数组合相混合的结果。幸运的是，我们大部分情况下可以对方法和函数一视同仁。

最后要介绍的是，偏函数与返回 Option 函数之间是可以相互转化的：

```
// src/main/scala/progscala2/fp/datastructs/lifted-func.sc

scala> val finicky: PartialFunction[String,String] = {
     |   case "finicky" => "FINICKY"
     | }
finicky: PartialFunction[String,String] = <function1>

scala> finicky("finicky")
res13: String = FINICKY
```

```
scala> finicky("other")
scala.MatchError: other (of class java.lang.String)
  ...

scala> val finickyOption = finicky.lift
finickyOption: String => Option[String] = <function1>

scala> finickyOption("finicky")
res14: Option[String] = Some(FINICKY)

scala> finickyOption("other")
res15: Option[String] = None

scala> val finicky2 = Function.unlift(finickyOption)
finicky2: PartialFunction[String,String] = <function1>

scala> finicky2("finicky")
res16: String = FINICKY

scala> finicky2("other")
scala.MatchError: other (of class java.lang.String)
  ...
```

这是函数提升的另一个用法。如果我们有一个偏函数，同时又不希望发生抛出异常的情况，可以将偏函数提升为一个返回 `Option` 的函数，也可以将返回 `Option` 的函数“降级”为偏函数。

## 6.7 函数式编程的数据结构

在面向对象语言中，我们往往会为各领域的概念建立一对一的类。而在函数式编程中，往往大量使用一些核心的数据结构和算法。尽管面向对象语言可以通过代码复用减少冗余，但类的膨胀还是非常明显。所以，尽管看上去有些矛盾，相比面向对象语言，函数式编程更倾向于写出简洁易复用的代码，因为函数式语言没有那么多的重复发明，它将重点放在使用核心数据结构和算法实现业务逻辑上。

不同的语言有不同的核心数据结构，但大致都包含同一个子集，子集中包含列表（list）、向量（vector）等序列型集合，数组（array），映射（map）与集合（set）。每种类型都支持同一批无副作用的高阶函数，称为*组合器*（combinator），如：`map`、`filter`、`fold` 等函数。一旦你了解了这些组合器，你就可以根据自身对数据访问的需要及性能要求，选用合适的集合类型，用同样的组合器函数去操作数据。在所有的软件开发工作中，这些集合类型是代码复用与组合的最有效工具。

### 6.7.1 序列

先来看几个在 Scala 编程中最常用的数据结构。

许多数据结构是序列型的，也就是说，元素可以按特定的顺序访问，如：元素的插入顺序或其他特定顺序。`collection.Seq`（http://www.scala-lang.org/api/current/#scala.collection.

Seq）是一个 trait，是所有可变或不可变序列类型的抽象，其子 trait `collection.mutable.Seq`（http://www.scala-lang.org/api/current/#scala.collection.mutable.Seq） 及 `collection.immutable.Seq`（http://www.scala-lang.org/api/current/#scala.collection.immutable.Seq） 分别对应可变和不可变序列。

列表也是一种序列，是函数式编程中最常用的数据结构，从第一代函数式语言 Lisp 就开始用列表了。

通常，向列表里追加元素时，该元素会被追加到列表的开头，成为新列表的“头部”。除了头部，剩下的部分就是原列表的元素，这些元素并没有被修改，它们变成了新列表的“尾部”。图 6-1 中有两个列表，`List(1,2,3,4,5)` 和 `List(2,3,4,5)`，后者是前者的尾部。

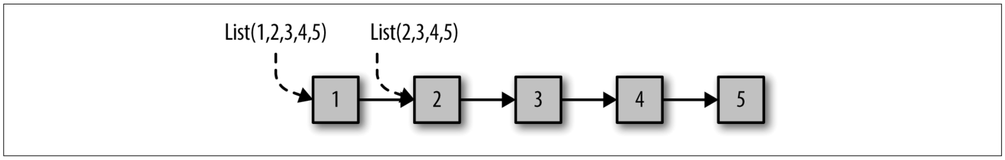



图 6-1：两个列表

我们从旧列表中创建新列表的操作是 $O(1)$。新列表的尾部与旧列表相同，只是在头部多了一个新元素。在函数式数据结构的思想中，我们将复制、共享数据结构的开销降到最低，这是第一个例子。为了支持可变性，我们必须要有降低复制开销的能力。

其他代码访问原队列时对新队列是无感知的。列表是不可变的，因此在另一个线程中，创建新队列的代码不会给访问旧列表的代码带来不可预料的修改操作。

从队列的创建过程，我们可以清楚地知道，计算队列长度的操作是 $O(N)$，其他需要从头遍历队列的操作也是如此。

以下 REPL 中的代码演示了如何在 Scala 中创建队列：

```
// src/main/scala/progscala2/fp/datastructs/list.sc

scala> val list1 = List("Programming", "Scala")
list1: Seq[String] = List(Programming, Scala)

scala> val list2 = "People" :: "should" :: "read" :: list1
list2: Seq[String] = List(People, should, read, Programming, Scala)
```

你可以用 `List.apply` 方法创建队列，然后用 `::` 方法（称为 cons，意为构造）向队列头部追加数据，从而创建新的列表。在这里我们使用了简单写法，省略了点号与小括号。我们提到过，以冒号（`:`）结尾的方法向右结合，因此 `x::list` 其实是 `list.::(x)`。

我们也可以用 `::` 方法向 `Nil` 空队列追加元素创建新队列：

```
scala> val list3 = "Programming" :: "Scala" :: Nil
list3: Seq[String] = List(Programming, Scala)

scala> val list4 = "People" :: "should" :: "read" :: Nil
list4: Seq[String] = List(People, should, read)
```

Nil 与 List.empty[Nothing] 是等价的，其中 Nothing 是 Scala 中所有其他类型的子类型。参见“Scala 类型层次”，了解更多关于 Nothing 的用法，你就会知道为什么在这里可以用它。

另外，你还可以用 ++ 方法将两个列表（或其他任何序列类型）连接起来：

```
scala> val list5 = list4 ++ list3
list5: Seq[String] = List(People, should, read, Programming, Scala)
```

在需要的时候构造列表的确是常用的做法，但不推荐方法将列表作为参数或作为返回值。而应该用 Seq，这样的话，Seq 的任何子类型就都可以用了，包括 List 和 Vector。

Seq 的构造方法是 +: 而不是 ::。以下是前文使用过的一个例子，其中用到了 Seq。注意，当你对伴随对象使用 Seq.apply 方法时，将创建出一个 List，这是因为 Seq 只是一个特征，而不是具体的类。

```
// src/main/scala/progscala2/fp/datastructs/seq.sc

scala> val seq1 = Seq("Programming", "Scala")
seq1: Seq[String] = List(Programming, Scala)

scala> val seq2 = "People" +: "should" +: "read" +: seq1
seq2: Seq[String] = List(People, should, read, Programming, Scala)

scala> val seq3 = "Programming" +: "Scala" +: Seq.empty
seq3: Seq[String] = List(Programming, Scala)

scala> val seq4 = "People" +: "should" +: "read" +: Seq.empty
seq4: Seq[String] = List(People, should, read)

scala> val seq5 = seq4 ++ seq3
seq5: Seq[String] = List(People, should, read, Programming, Scala)
```

在这里，我们用 Seq.empty 为 seq3 和 seq4 创建了空的队列作为队尾。在 Scala 中，大部分集合类型的伴随对象都使用 empty 方法来创建该类型的空实例，类似列表的 Nil 实例。

序列类型还定义了 :+ 和 +: 方法。它们有什么区别，怎么记住哪个是哪个呢？只要记住 : 总是靠近集合类型就可以了，比如：list :+ x，x +: list。所以，:+ 方法用于在尾部追加元素，+: 方法用于在头部追加元素：

```
scala> val seq1 = Seq("Programming", "Scala")
seq1: Seq[String] = List(Programming, Scala)

scala> val seq2 = seq1 :+ "Rocks!"
seq2: Seq[String] = List(Programming, Scala, Rocks!)
```

Scala 定义的 List 是不可变的，不过，它还定义了其他可变的列表类型，如 ListBuffer（http://www.scala-lang.org/api/current/scala/collection/mutable/ListBuffer.html） 和 MutableList（http://www.scala-lang.org/api/current/scala/collection/mutable/MutableList.html）。只有当必须修改元素时才可以使用可变类型。

虽然 List 对于序列类型是个不错的选择，你也可以考虑用 immutable.Vector 代替 List，因为 immutable.Vector（http://www.scala-lang.org/api/current/scala/collection/immutable/

Vector.html）的所有操作都是 *O*(1)（常数时间），而 `List` 对于那些需要访问头部以外元素的操作，都需要 *O*(*n*) 操作。

以下例子是 `Vector` 对之前例子的重写。除了将 `Seq` 换为 `Vector`，并修改了变量名以外，其余代码完全相同：

```
// src/main/scala/progscala2/fp/datastructs/vector.sc

scala> val vect1 = Vector("Programming", "Scala")
vect1: scala.collection.immutable.Vector[String] = Vector(Programming, Scala)

scala> val vect2 = "People" +: "should" +: "read" +: vect1
vect2: ...Vector[String] = Vector(People, should, read, Programming, Scala)

scala> val vect3 = "Programming" +: "Scala" +: Vector.empty
vect3: ...Vector[String] = Vector(Programming, Scala)

scala> val vect4 = "People" +: "should" +: "read" +: Vector.empty
vect4: ...Vector[String] = Vector(People, should, read)

scala> val vect5 = vect4 ++ vect3
vect5: ...Vector[String] = Vector(People, should, read, Programming, Scala)
```

我们能够以常数时间复杂度获取任意元素：

```
scala> vect5(3)
res0: String = Programming
```

在结束对序列的讨论之前，你还需要知道一个实现上的细节问题。为了鼓励程序员使用不可变的集合类型，`Predef` 及 `Predef` 中使用的其他类型在暴露部分不可变集合类型时，不需要显式导入或使用全路径导入。如：`List` 和 `Map`。

在上述规则中，Scala 只暴露了不可变集合类型，然而，`Predef` 还将 `scala.collection.Seq` 导入到了当前作用域。`scala.collection.Seq` 中的类型是可变集合类型和不可变集合类型共同的抽象类型。

尽管存在 `scala.collection.immutable.Seq`（`scala.collection.Seq` 的一个子类型），但 `Predef` 导入的是 `scala.collection.immutable.Seq` 而非 `scala.collection.Seq`，主要原因是这方便处理 Java 的 `Array`。如同 `Seq` 一样，`Array` 和其他集合类型有着相同的处理方式。Java 的 `Array` 是可变集合类型，Scala 的其他可变集合类型也大部分都实现了 `scala.collection.Seq`。

这样一来，`scala.collection.Seq` 虽然没有暴露任何用于修改集合的方法，但仍存在并发情况下出错的潜在风险，因为可变集合类型不是线程安全的，因此必须对它做特殊处理。

假设并发库中的方法将 `Seq` 作为参数，但你只希望传入不可变集合类型。此时 `Seq` 的使用就产生了一个漏洞，因为客户端可以传入可变集合类型，如 `Array`。

> 应该记住，`Seq` 默认的实际类型为 `scala.collection.Seq`。因此，传入的 `Seq` 类型的实例可能是可变的，所以线程是不安全的。

Scala 计划在 2.12 版本（即下一个发行版）中将 scala.Seq 改为 scala.collection.immutable.Seq 的指向别名。

在那之前，如果你真的想用 scala.collection.immutable.Seq，可以用下文所示的技术（改编自 Heiko Seeberger 的博客，http://hseeberger.github.io/blog/2013/10/25/attention-seq-is-not-immutable/）。

我们将在 2.12.2 节介绍包对象。运用包对象，我们为 Seq 定义新的别名，以覆盖原有的别名定义。事实上，Seq 就是在包对象 scala 中被定义为 scala.collection.Seq 的别名。在本节中我们已经用过 fp.datastructs，接下来我们将在这个包中定义包对象：

```
// src/main/scala/progscala2/fp/datastructs/package.scala
package progscala2.fp
package object datastructs {
  type Seq[+A] = scala.collection.immutable.Seq[A]
  val Seq = scala.collection.immutable.Seq
}
```

注意到，在包对象 datastructs{…} 的定义之前，我们指定的是包 fp，而不是包 fp.datastructs。package 关键字也是包对象定义的一部分，在一个包中只能有一个包对象。最后，我们要注意注释中该文件的路径，文件名为 package.scala，这是一种常用的命名习惯。

在包对象中，我们声明了一个类型别名和一个 val 变量。关于类型声明，可以回顾抽象类型与参数化类型。如果用户代码中包含了导入语句 import fp.datastructs._，那么当使用 Seq（不带包修饰符）声明一个实例时，就需要使用 scala.collection.immutable.Seq 代替默认的 scala.collection.Seq。

val Seq 的声明语句将伴随对象引入作用域，于是类似 Seq(1,2,3,4) 的语句将会触发 scala.collection.immutable.Seq.apply 方法的调用。

在 fp.datastructs 下的包如何处理呢？如果你要在该层级中实现一个包，可以使用我们在 2.11 节讨论的包继承语句：

```
package fp.datastructs     // 使得Seq指向immutable.Seq
package asubpackage        // 包中的内容
package asubsubpackage     // 我正在开发的包
```

这种在包对象中定义类型别名的方法，可以用来暴露你自己定义的 API 中的最重要类型。

## 6.7.2 映射表

另一种常用的数据结构是映射（http://www.scala-lang.org/api/current/scala/collection/immutable/Map.html），在其他语言中有时也被称为散列、散列表。映射表用来存储键值对，但不应将其与很多数据结构的 map 方法混淆。映射表与 map 方法有一定程度的类似，前者每个键都对应一个值，后者每个输入元素都产生一个输出元素。

如前文所述，Scala 支持对映射表采用以下特殊的初始化语法：

```
// src/main/scala/progscala2/fp/datastructs/map.sc
```

```
scala> val stateCapitals = Map(
     |   "Alabama" -> "Montgomery",
     |   "Alaska"  -> "Juneau",
     |   "Wyoming" -> "Cheyenne")
stateCapitals: scala.collection.immutable.Map[String,String] =
  Map(Alabama -> Montgomery, Alaska -> Juneau, Wyoming -> Cheyenne)

scala> val lengths = stateCapitals map {
     |   kv => (kv._1, kv._2.length)
     | }
lengths: scala.collection.immutable.Map[String,Int] =
  Map(Alabama -> 10, Alaska -> 6, Wyoming -> 8)

scala> val caps = stateCapitals map {
     |   case (k, v) => (k, v.toUpperCase)
     | }

caps: scala.collection.immutable.Map[String,String] =
  Map(Alabama -> MONTGOMERY, Alaska -> JUNEAU, Wyoming -> CHEYENNE)

scala> val stateCapitals2 = stateCapitals + (
     |   "Virginia" -> "Richmond")
stateCapitals2: scala.collection.immutable.Map[String,String] =
  Map(Alabama -> Montgomery, Alaska -> Juneau,
  Wyoming -> Cheyenne, Virginia -> Richmond)

scala> val stateCapitals3 = stateCapitals2 + (
     |   "New York" -> "Albany", "Illinois" -> "Springfield")
stateCapitals3: scala.collection.immutable.Map[String,String] =
  Map(Alaska -> Juneau, Virginia -> Richmond, Alabama -> Montgomery,
  New York -> Albany, Illinois -> Springfield, Wyoming -> Cheyenne)
```

在前文中我们已经了解到，`key -> value` 的语法形式实际上是用库中的隐式转换实现的，实际调用了 `Map.apply` 方法。`Map.apply` 方法的参数为一个两元素的元组（键值对）。

`map` 的参数是一个函数，以上示例展示了定义这个函数的两种方法。每个键值对（元组）都将被传入到该函数中。我们可以将其参数定义为一个两元素的元组，或使用类似第 4 章中讨论过的 `case` 语法从元组中提取键和值。

最后，我们可以用 + 向 `Map` 中添加一个或多个键值对（来创建新的 `Map` 实例）。

顺便提一下，如果我们在 `stateCapitals + ("Virginia" -> "Richmond")` 语句中漏掉了小括号，那会发生什么呢？

```
scala> stateCapitals + "Virginia" -> "Richmond"
res2: (String, String) = (Map(Alabama -> Montgomery, Alaska -> Juneau,
  Wyoming -> Cheyenne)Virginia,Richmond)
```

什么？我们得到了字符串对 `(String,String)`。不幸的是，Scala 会在必要的时候将其他类型转为 `String`。由于没有其他更好的选择，+ 方法被用于连接两个字符串。编译器首先执行 `stateCapitals.toString + "Virginia"`，产生一个字符串；然后执行 `->`，创建字符串元组，其中 Richmond 是该元组的第二个元素。于是，结果为元组 `("Map(Alabama -> … ->`

Cheyenne)Virginia", "Richmond")。

> 如果一个包含 + 的表达式返回结果类型 String，而这并不是你所期望的结果。这可能是因为编译器认为这是该表达式唯一可行的解析方式，就把 + 两边的子表达式转为字符串，再相加。

实际上，我们并不能调用 new Map("Alabama" -> "Montgomery", …) 方法，因为它是一个 trait。相反，Map.apply 则会根据给定的输入数据用最佳的方式构造实例。Map.apply 通常根据键值对个数来构造实例，例如：构造出包含一个、两个、三个和四个键值对的映射表。

与 List 不同，Map 有可变和不可变两种实现：分别是 scala.collection.immutable.Map[A,B] 与 scala.collection.mutable.Map[A,B]。可变的实现需要显式导入，不可变的实现则已经用 Predef 暴露出来了。两种实现都定义了 + 和 - 操作用于增加和移除元素；以及 ++ 和 -- 操作来增加和移除 Iterator 中定义的元素（Iterator 也可以换为其他集合、列表等）。

### 6.7.3 集合

集合是无序集合类型的一个例子，所以集合不是序列。集合同样要求元素具有唯一性：

```
// src/main/scala/progscala2/fp/datastructs/set.sc
scala> val states = Set("Alabama", "Alaska", "Wyoming")
states: scala.collection.immutable.Set[String] = Set(Alabama, Alaska, Wyoming)

scala> val lengths = states map (st => st.length)
lengths: scala.collection.immutable.Set[Int] = Set(7, 6)

scala> val states2 = states + "Virginia"
states2: scala.collection.immutable.Set[String] =
  Set(Alabama, Alaska, Wyoming, Virginia)

scala> val states3 = states2 + ("New York", "Illinois")
states3: scala.collection.immutable.Set[String] =
  Set(Alaska, Virginia, Alabama, New York, Illinois, Wyoming)
```

类似 Map，特征 scala.collection.Set（http://www.scala-lang.org/api/current/scala/collection/Set.html）只定义不可变操作的方法。对于具体的不可变集合和可变集合，分别定义派生的特征 scala.collection.immutable.Set（http://www.scala-lang.org/api/current/scala/collection/immutable/Set.html）和 scala.collection.mutable.Set（http://www.scala-lang.org/api/current/scala/collection/mutable/Set.html）。可变的版本需要显式导入，而不可变的版本在 Predef 中已经导入了。两者都为增加和移除元素定义了 + 和 - 操作；为 Iterator（也可以为其他集合、列表等）中的增加和移除元素定义了 ++ 和 -- 操作。

## 6.8 遍历、映射、过滤、折叠与归约

常见的集合类型——序列、列表、集合、数组、树及其他类似的类型，都支持基于只读遍

历的通用操作。事实上，这一标准性可以被充分利用起来，特别是你实现的某个“容器”类型也支持这些操作的情况。例如 Option 是包含零个 None 或一个 Some 元素的容器。

### 6.8.1 遍历

Scala 容器类型的标准遍历方法是 foreach，foreach 定义于 scala.collection.IterableLike（http://www.scala-lang.org/api/current/index.html#scala.collection.IterableLike?）中，它的签名为：

```
trait IterableLike[A] {  // 省略部分细节
  ...
  def foreach[U](f: A => U): Unit = {...}
  ...
}
```

IterableLike 的部分子类型可能会重新定义该方法，以利用程序的本地信息，获得更好的性能。

在上述代码中，U 是函数 f 的返回类型，事实上 U 具体是什么类型并不重要。foreach 函数的输出类型是 Unit，因此 foreach 是一个完全副作用的函数。又因为它的参数是一个函数，foreach 又是一个高阶函数。注意这里讨论的所有操作函数均为高阶函数。

foreach 的复杂度是 $O(N)$，$N$ 为元素个数。以下为 foreach 在列表和映射表中的用法：

```
// code-examples/progscala2/fp/datastructs/foreach.sc

scala> List(1, 2, 3, 4, 5) foreach { i => println("Int: " + i) }
Int: 1
Int: 2
Int: 3
Int: 4
Int: 5

scala> val stateCapitals = Map(
     |   "Alabama" -> "Montgomery",
     |   "Alaska"  -> "Juneau",
     |   "Wyoming" -> "Cheyenne")
stateCapitals: scala.collection.immutable.Map[String,String] =
  Map(Alabama -> Montgomery, Alaska -> Juneau, Wyoming -> Cheyenne)

// stateCapitals foreach { kv => println(kv._1 + ": " + kv._2) }

scala> stateCapitals foreach { case (k, v) => println(k + ": " + v) }
Alabama: Montgomery
Alaska: Juneau
Wyoming: Cheyenne
```

注意，在映射表的例子中，传递给 foreach 的 A 参数实际上是一个键值对 (K,V)。我们用模式匹配表达式将键和值提取了出来，注释中展示的是另一种直接使用元组的合法写法 [2]。

---

注 2：匿名函数中运用 case 语句，实际上定义了一个偏函数，但这个函数实际上并不“偏”，因为它可以匹配所有输入。

foreach 并不是一个纯函数，因为 foreach 只能执行带副作用的操作。然而，一旦有了 foreach，我们就可以实现其他接下来要讨论的不带副作用的操作。这些操作是函数值编程的标志：映射、过滤、折叠、归约。

### 6.8.2 映射

我们之前已经接触过 map 方法，map 方法返回一个与原集合类型大小相同的新集合，其中的每个元素均由原集合的对应元素转换得到。map 方法被定义于 scala.collection.TraversableLike（http://www.scala-lang.org/api/current/index.html#scala.collection.TraversableLike），被大部分集合类型继承。其签名如下：

```
trait TraversableLike[A] {  // 省略部分细节
  ...
  def map[B](f: (A) ⇒ B): Traversable[B]
  ...
}
```

在本章上文中，我们已经接触过关于映射的实例了。

事实上，map 的这个方法签名并不是 map 在源代码中的签名，而是显示在 Scaladoc 中的签名。如果你阅读 Scaladoc（http://www.scala-lang.org/api/current/index.html#scala.collection.TraversableLike），会在结尾找到一个“完整”的方法签名，点击附近的箭头可以将其展开显示。这个签名才是 map 方法的真正签名：

```
def map[B, That](f: A => B)(implicit bf: CanBuildFrom[Repr, B, That]): That
```

最近版本的 Scaladoc 为了揭示方法的本质，使读者不被淹没在方法的实现细节中，对一部分方法的签名做了简化。map 方法的实质是将一个集合类型转换为一个类型相同、大小相同的新集合，而其中原集合 A 的元素经过转换函数处理，转为集合 B 中的元素。

map 方法的真正签名包含了实现的细节信息。That 是输出集合的类型，事实上与输入集合的类型相同。我们在 5.2.2 节讨论过隐含参数 bf: CanBuildFrom 的作用。它的存在意味着我们可以用 map 的输出和函数 f 构造一个 That，同时 bf: CanBuildFrom 本身也负责构建。Repr 是内部用来表示集合元素的。

尽管隐含参数 CanBuildFrom 增加了方法签名的复杂性（还引来了邮件列表里的诸多抱怨），但如果没有它，我们就无法创建新的 Map、List 或其他继承自 TraversableLike 的集合。这是使用面向对象继承层次模型来实现集合 API 的自然结果。所有具体的集合类型都可以重新实现 map 方法，并返回一个该类型的新实例。但这种放弃代码重用的做法将削弱使用面向对象带来的好处。

事实上，第 1 版面世时，普遍使用的是 Scala v2.7.X，在该版本中集合类型的 API 并没有以上特性。map 和其他方法返回一个 Iteratable 或类似的抽象类型，这个类型往往只是 Array 或其他底层类型。

从现在开始，本书将只给出方法的简化签名，以集中精力研究方法的行为特性。不过，我鼓励大家选择一个集合类型，如 List 或 Map，然后阅读各个方法的完整签名形式。刚开始阅读这些方法签名时可能会令人气馁，但这对有效地使用这些集合来说是必要的。

另一个关于 map 的有趣事实也值得关注。再来看一下 map 方法的简化签名。为了方便，这里只拿 List 作为示例，对这个例子来说用哪个集合都一样：

```
trait List[A] {
  ...
  def map[B](f: (A) ⇒ B): List[B]
  ...
}
```

参数是一个函数 A => B，而事实上，map 方法的行为是 List[A] => List[B]。这一点往往被对象语法如 list map f 所掩盖。如果我们有其他函数模块使用 List 作为参数，函数形式会是这样。

```
// src/main/scala/progscala2/fp/combinators/combinators.sc

object Combinators1 {
  def map[A,B](list: List[A])(f: (A) ⇒ B): List[B] = list map f
}
```

（我作弊用了 List.map 来实现这个函数……）

如果我们交换参数列表的顺序会如何？

```
object Combinators {
  def map[A,B](f: (A) ⇒ B)(list: List[A]): List[B] = list map f
}
```

最后，我们在 REPL 中查看这个函数：

```
scala> object Combinators {
     |   def map[A,B](f: (A) ⇒ B)(list: List[A]): List[B] = list map f
     | }
defined module Combinators

scala> val intToString = (i:Int) => s"N=$i"
intToString: Int => String = <function1>

scala> val flist = Combinators.map(intToString) _
flist: List[Int] => List[String] = <function1>

scala> val list = flist(List(1,2,3,4))
list: List[String] = List(N=1, N=2, N=3, N=4)
```

关键在于第二步和第三步。在第二步中，我们定义了类型为 Int => String 的函数 intToString。intToString 对 List 一无所知。在第三步中，我们用 Combinators.map 定义了一个新函数，flist 的类型是 List[Int] => List[String]。所以，我们用 map 将一个类型为 Int => String 的函数提升为类型是 List[Int] =>List[String] 的函数。

通常，我们认为 map 方法总是将元素类型为 A 的集合转为大小相同但元素类型为 B 的集合，使用的转换函数 f: A => B 对该集合一无所知。现在我们知道，map 也可以被看作是一个将函数 f: A => B 提升为新函数 flist: List[A] => List[B] 的工具。在例子中我们用的是列表，但这对有 map 方法的任何容器类型均适用。

不幸的是，我们并不能直接用 Scala 标准库的 map 方法达成这一目的，因为 map 是实例方法。所以，我们无法用它来为所有的 List 实例创建提升函数。

也许这是 Scala 采用面向对象和函数式混合范式的结果。但这个用法的使用场景并不多见。大部分时候，你会有一个集合类型的实例，然后你可以指定一个函数参数来调用 map 方法，以创建一个新的集合实例。希望将一个普通函数提升为操作集合的函数，输入一个集合实例可以输出一个新实例，这种应用并不常见。

### 6.8.3 扁平映射

flatMap 是 Map 操作的一种推广。在 flatMap 中，我们对原始集合中的每个元素，都分别产生零或多个元素。我们传入一个函数，该函数对每个输入返回一个集合，而不是一个元素。然后 flatMap 把生成的多个集合“压扁”为一个集合。

以下就是 TraversableLike 中 flatMap 方法的简化版签名，同时还给出了 map 方法的签名作为对比：

```
def flatMap[B](f: A => GenTraversableOnce[B]): Traversable[B]
def map[B](f: (A) => B): Traversable[B]
```

注意，map 中函数 f 的签名为 A => B。现在我们需要返回一个集合，GenTraversableOnce 就是这样一个接口，它表示可至少遍历一次的任何实体。

考虑以下这个例子：

```
// src/main/scala/progscala2/fp/datastructs/flatmap.sc

scala> val list = List("now", "is", "", "the", "time")
list: List[String] = List(now, is, "", the, time)

scala> list flatMap (s => s.toList)
res0: List[Char] = List(n, o, w, i, s, t, h, e, t, i, m, e)
```

对每个字符串调用 toList，生成 List[Char]。这些嵌套的列表随后被“压扁”为最终的 List[Char]。变量 list 中的空字符串对最终生成的字符串没有贡献，但其他字符串却分别共享了两个或更多字符。

事实上，flatMap 的行为很像先调用 map，再调用另一个方法 flatten：

```
// src/main/scala/progscala2/fp/datastructs/flatmap.sc

scala> val list2 = List("now", "is", "", "the", "time") map (s => s.toList)
list2: List[List[Char]] =
  List(List(n, o, w), List(i, s), List(), List(t, h, e), List(t, i, m, e))

scala> list2.flatten
res1: List[Char] = List(n, o, w, i, s, t, h, e, t, i, m, e)
```

在这里，起媒介作用的临时变量是集合 List[List[Char]]。然而，flatMap 比连续调用以上两个方法更高效，因为 flatMap 不需要创建临时变量。

要注意的是，flatMap 不能处理超过一层的集合。如果函数返回的是深层嵌套的集合，那么集合只能被压扁一层。

## 6.8.4 过滤

遍历一个集合，然后抽取其中满足特定条件的元素，组成一个新的集合，这是一种很常见的需求：

```
// src/main/scala/progscala2/fp/datastructs/filter.sc

scala> val stateCapitals = Map(
     |   "Alabama" -> "Montgomery",
     |   "Alaska"  -> "Juneau",
     |   "Wyoming" -> "Cheyenne")
stateCapitals: scala.collection.immutable.Map[String,String] =
  Map(Alabama -> Montgomery, Alaska -> Juneau, Wyoming -> Cheyenne)

scala> val map2 = stateCapitals filter { kv => kv._1 startsWith "A" }
map2: scala.collection.immutable.Map[String,String] =
  Map(Alabama -> Montgomery, Alaska -> Juneau)
```

在 scala.collection.TraversableLike 中有若干不同的方法，用于完成集合的过滤作用或者返回原始集合中的一部分元素。一些方法在输入无限集合时不会返回；一些方法在输入同一个集合时，除非集合的遍历顺序固定，否则多次运行的情况下会产生不同的输出。以下是从 Scaladoc 中摘录的介绍。

- `def drop (n : Int) : TraversableLike.Repr`

  除起始的 *n* 个元素，选择其他所有的元素组成一个新的集合并返回。如果原始集合包含的元素个数小于 *n*，则方法会返回一个空集合。

- `def dropWhile (p : (A) => Boolean) : TraversableLike.Repr`

  从头遍历，丢弃满足一定谓词的最长集合前缀。返回一个最长集合后缀，其第一个元素不满足指定的谓词 p。

- `def exists (p : (A) => Boolean) : Boolean`

  测试在集合中是否至少有一个元素满足给定的谓词，如果存在则返回 true，否则返回 false。

- `def filter (p : (A) => Boolean) : TraversableLike.Repr`

  选择集合中所有满足一定谓词的元素，返回的新集合中包含了所有满足该谓词 p 的元素。元素在原集合中的顺序可以得到保持。

- `def filterNot (p : (A) => Boolean) : TraversableLike.Repr`

  是 filter 的“反义词”。在遍历原集合时，选择那些不满足给定谓词 p 的元素并组成新集合返回。

- `def find (p : (A) => Boolean) : Option[A]`

  遍历原集合，寻找第一个满足给定谓词的元素。如果存在这一元素，返回 Option，且 Option 中包含满足谓词 p 的第一个元素；否则返回 None。

- `def forall (p : (A) => Boolean) : Boolean`

  测试集合中所有元素是否均满足给定的谓词。如果所有元素均满足谓词 `p`，则返回 true，否则返回 false。

- `def partition (p : (A) => Boolean): (TraversableLike.Repr, TraversableLike.Repr)`

  根据谓词，将可遍历集合分成两个子集合。返回值是两个集合：第一个集合包含所有满足谓词 `p` 的元素，而第二个集合包含所有不满足谓词 `p` 的元素。两个集合中元素间的顺序均与原集合保持一致。

- `def take (n : Int) : TraversableLike.Repr`

  选择前 *n* 个元素。返回一个可遍历集合，包含原集合的前 *n* 个元素，如果原集合包含的元素小于 *n* 个，则返回原集合本身。

- `def takeWhile (p : (A) => Boolean) : TraversableLike.Repr`

  选择满足特定谓词的最长集合前缀。返回的可遍历集合包含一个最长集合前缀，其中的每个元素均满足谓词 `p`。

另外，很多集合类型还有其他用于过滤的方法。

## 6.8.5　折叠与归约

我们把折叠和归约放在一起讨论是因为两者很相似。它们都是将一个集合“缩小”成一个更小的集合或一个值的操作。

折叠从一个初始的“种子”值开始，然后以该值作为上下文，处理集合中的每个元素。不同的是，归约不需要调用者提供一个初始值。它将集合的其中一个元素当做初始值，通常这个值是集合的第一个元素或最后一个元素：

```
// src/main/scala/progscala2/fp/datastructs/foldreduce.sc

scala> val list = List(1,2,3,4,5,6)
list: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> list reduce (_ + _)
res0: Int = 21

scala> list.fold (10) (_ * _)
res1: Int = 7200

scala> (list fold 10) (_ * _)
res1: Int = 7200
```

以上脚本通过累加各个元素，将整数列表归约一个整数值，返回值为 21。脚本接着用 10 作为初始值，对同一个列表的元素做累乘，得到返回值 7200。

与 `reduce` 类似，传给 `fold` 的函数需要两个参数，包括累计值和初始值。新的累计值由该函数计算得到。在这个例子中，我们对累计值和元素依次做加法或乘法，以得到新的累计值。

以上示例给出了两种书写 `fold` 表达式的方法，两种方法都需要两个参数列表：初始种子值和用于计算结果的累计函数。所以，我们无法使用 `reduce` 中采用的中缀表达法。

不过，如果我们真的想采用中缀表达法的话，我们可以像 (list fold 10) 一样使用括号，括号后面跟上表示函数的参数列表。我本人更倾向于这种语法。

可以用括号解决这种问题的原因并不那么明显。为了揭示其工作机理，考虑如下代码：

```
scala> val fold1 = (list fold 10) _
fold1: ((Int, Int) => Int) => Int = <function1>

scala> fold1(_ * _)
res10: Int = 7200
```

我们用偏应用的方法创建了 fold1，随后我们给出了剩下的参数列表 (_ * _) 以调用 fold1。能够想到 (list fold 10) 是偏应用的开头，随后加上后面的函数 (_ * _)。

如果我们对任何一个空的集合执行 fold 操作，它会返回初始种子的值。与此不同，reduce 无法对空集合进行操作，因为 reduce 没有值可以返回。如果那样，就会抛出异常：

```
scala> (List.empty[Int] fold 10) (_ + _)
res0: Int = 10

scala> List.empty[Int] reduce (_ + _)
java.lang.UnsupportedOperationException: empty.reduceLeft
...
```

然后，如果你不确定集合是否为空（例如：当集合是传入到你函数中的一个参数时），你可以用 optionReduce 代替 reduce：

```
scala> List.empty[Int] optionReduce (_ + _)
res1: Option[Int] = None

scala> List(1,2,3,4,5) optionReduce (_ + _)
res2: Option[Int] = Some(15)
```

reduce 返回集合中各元素的最近公共父类型[3]。如果元素均为同一类型,则 reduce 返回的元素就是该类型的。fold 方法则有初始种子值，因为对最终结果的处理有更多选项。以下是一个折叠操作，事实上相当于映射操作：

```
// src/main/scala/progscala2/fp/datastructs/fold-map.sc

scala> (List(1, 2, 3, 4, 5, 6) foldRight List.empty[String]) {
     |   (x, list) => ("[" + x + "]") :: list
     | }
res0: List[String] = List([1], [2], [3], [4], [5], [6])
```

首先，我们使用 fold 的一个变体 foldRight 从右到左遍历集合，这样可以保证我们构造的新序列中元素的顺序是正确的。也就是说，元素 6 首先被处理，并插入到空序列头部；随后元素 5 被处理，插入到该序列的头部，以次类推。这样，累计值相当于这个匿名函数的第二个参数。

事实上，所有其他的操作均可用 fold 实现，包括 foreach。如果你只能拥有我们讨论的众多函数中的一个，那么你可以选择 fold，然后用 fold 重新实现其他函数。

注 3：也称为最小上界或 LUB。

以下我们就给出关于 fold 和 reduce 方法的签名和介绍，它们被声明于 scala.collection.TraversableOnce 和 scala.collection.TraversableLike 中。下面的介绍是从 Scaladoc 中摘录的。

- def fold[A1 >: A](z: A1)(op: (A1, A1) ⇒ A1): A1

  遍历集合，使用指定的二元操作符 op 对集合元素做折叠。对元素执行操作时，其顺序是未指定的，因此结果是不能确定的。不过，对于多数顺序固定的集合如 List，fold 的作用与 foldLeft 相同。

- def foldLeft[B](z: B)(op: (B, A) ⇒ B): B

  对初始值和集合中的所有元素做操作，顺序从左到右。

- def foldRight[B](z: B)(op: (A, B) ⇒ B): B

  对初始值和集合中的所有元素做操作，顺序从右到左。

- def /:[B](z: B)(op: (B, A) ⇒ B): B = foldLeft(z)(op)

  foldLeft 的别名。调用形式例如：(0 /: List(1,2,3))(_ + _)。不过，大部分人认为用操作符 /: 来表示 foldLeft 太隐晦不易记忆，所以，写代码时不要忘记与代码阅读者进行交流。

- def :\[B](z: B)(op: (A, B) ⇒ B): B = foldRight(z)(op)

  foldRight 的别名。调用形式例如：(0 /: List(1,2,3))(_ + _)。不过，大部分人认为用操作符 /: 来表示 foldLeft 太隐晦不易记忆。

- def reduce[A1 >: A](op: (A1, A1) ⇒ A1): A1

  遍历集合，使用指定的二元操作符 op 对集合元素做归约。对元素执行操作时，其顺序是未指定的，因此结果是不能确定的。不过，对于多数顺序固定的集合如 List，reduce 的作用与 reduceLeft 相同。如果集合为空，则抛出异常。

- def reduceLeft[A1 >: A](op: (A1, A1) ⇒ A1): A1

  对初始值和集合中的所有元素做操作，顺序从左到右。如果集合为空，则抛出异常。

- def reduceRight[A1 >: A](op: (A1, A1) ⇒ A1): A1

  对初始值和集合中的所有元素做操作，顺序从右到左。如果集合为空，则抛出异常。

- def optionReduce[A1 >: A](op: (A1, A1) ⇒ A1): Option[A1]

  类似 reduce，但当集合为空时，返回 None；集合不空时，返回 Some(…)。

- def reduceLeftOption[B >: A](op: (B, A) ⇒ B): Option[B]

  类似 reduceLeft，但当集合为空时，返回 None；集合不空时，返回 Some(…)。

- def reduceRightOption[B >: A](op: (B, A) ⇒ B): Option[B]

  类似 reduceRight，但当集合为空时，返回 None；集合不空时，返回 Some(…)。

- def aggregate[B](z: B)(seqop: (B, A) ⇒ B, combop: (B, B) ⇒ B): B

  对后面的元素进行操作，并聚合结果。该函数比 fold 和 reduce 形式更加通用，具有相似的语法，但并不要求结果的类型是元素的公共父类型。函数将集合分为不同的分片，

顺序遍历各个分片，用 seqop 更新计算结果，最后对各个分片的计算结果调用 combop。该操作可能操作任意个分片，因此 combop 可能被调用任意次。

- `def scan[B >: A](z: B)(op: (B, B) ⇒ B): TraversableOnce[B]`

  扫描、计算集合元素的一个前缀。中间的元素 z 可能会被多次操作。（在本节末尾，会给出一个示例。）

- `def scanLeft[B >: A](z: B)(op: (B, B) ⇒ B): TraversableOnce[B]`

  从左到右遍历集合，对元素执行 op 操作，得到一个包含一系列累计值的集合。

- `def scanRight[B >: A](z: B)(op: (B, B) ⇒ B): TraversableOnce[B]`

  从右到左遍历集合，对元素执行 op 操作，得到一个包含一系列累计值的集合。

- `def product: A`

  计算集合元素的累乘值。只要集合元素可以隐式转换为 Numeric[A]（http://www.scala-lang.org/api/current/scala/math/Numeric.html）（例如：Int、Long、Float、Double 和 BigInt），就可以返回元素的累乘值。实际上，函数的完整签名为 `def product[B >: A](implicit num: Numeric[B]): B`，参见 5.2.3 节，了解关于使用隐式转换来约束方法使用范围的细节。

- `def mkString: String`

  将集合中的所有元素序列化到字符串中。该方法是 fold 的一个自定义实现，用于方便地从集合生成字符串。在集合的元素之间没有分隔符。

- `def mkString(sep: String): String`

  从集合生成字符串，分隔符为字符串参数 sep。

- `def mkString(start: String, sep: String, end: String): String`

  从集合生成字符串，分隔符为字符串参数 sep，前缀为 start，后缀为 end。

特别留心传递给 reduce、fold 和 aggregate 的匿名函数的参数。对于 Left 系列的函数，如 foldLeft，其中第一个参数为累计值，集合遍历的方向为从左向右。对于 Right 系列的函数，如 foldRight，其中第二个参数为累计值，集合遍历的方向为从左向右。对于 fold 和 reduce 等方法，遍历方向既不是从左到右，也不是从右到左，其遍历方向与初始值是未定义的（不过一般都采用了从左到右的方式）。

fold 系列方法可以输出一个与集合元素完全不同类型的值；而 reduce 系列方法总是返回与元素相同类型或元素父类型的值。

以上方法对于无限集合均不终止处理。同时，如何集合不是序列类型（非序列类型中，元素的存储顺序不固定）或操作不满足交换律的，以上方法每次运行返回的结果可能都不相同。

aggregate 方法的应用并不广泛，因为它包含一些比较难以理解的“不固定成分”。

scan 方法对计算集合的连续子集非常有用。考虑以下例子：

```
scala> val list = List(1, 2, 3, 4, 5)
list: List[Int] = List(1, 2, 3, 4, 5)

scala> (list scan 10) (_ + _)
res0: List[Int] = List(10, 11, 13, 16, 20, 25)
```

首先，输出初始值 10，接下来输出第一个元素与初始值的和 11，然后是第二个元素与前值的和，11 + 2 = 13，以此类推。

最后，我们介绍了 3 个 mkString 方法，因为它们事实上是 fold 和 reduce 的特例，用于生成 String。如果集合默认的 toString 方法并不是你想要的，这些方法仍然十分有用。

## 6.9 向左遍历与向右遍历

从上一节可知，fold 和 reduce 函数不保证固定的遍历顺序。而 foldLeft 和 reduceLeft 则从左到右遍历集合元素，foldRight 和 reduceRight 则从右到左遍历集合元素。

这样，任何需要保持集合原有顺序的操作（例如，将整数列表转为格式化字符串的列表），就必须使用 foldLeft 或 foldRight 才行。

这一节我们就来讨论 fold 和 reduce 的向左遍历和向右遍历这两种形式，它们在行为上有很重要的区别。

我们再次列出上文所用的例子，不过现在用 fold、foldLeft、foldRight、reduce、reduceLeft 和 reduceRight 作为对比。首先，给出 fold 的例子：

```
scala> (List(1,2,3,4,5) fold 10) (_ * _)
res0: Int = 1200

scala> (List(1,2,3,4,5) foldLeft 10) (_ * _)
res1: Int = 1200

scala> (List(1,2,3,4,5) foldRight 10) (_ * _)
res2: Int = 1200
```

然后是 reduce 的例子：

```
scala> List(1,2,3,4,5) reduce (_ + _)
res3: Int = 15

scala> List(1,2,3,4,5) reduceLeft (_ + _)
res4: Int = 15

scala> List(1,2,3,4,5) reduceRight (_ + _)
res5: Int = 15
```

好了，不是特别令人振奋，因为选择不同的方法似乎对结果并没有产生影响。原因在于我们使用的匿名函数 _ * _ 和 _ + _ 满足交换律和结合律。

进一步分析。首先，对于 List，fold 只是调用给了 foldLeft，因此它们表现出相同的行为。这在大部分情况下是对的，但并不是对所有的集合都是如此。所以，我们集中关注 foldLeft 和 foldRight。其次，foldLeft 和 foldRight 采用了相同的匿名函数，但两次调

用时，参数事实上是相反的。在 foldLeft 中，匿名函数的第一个参数是累计值，而对于 foldRight，第二个参数才是累计值。

我们用满足交换律和结合律的函数来比较 foldLeft 和 foldRight，以便使得参数更清晰：

```
// src/main/scala/progscala2/fp/datastructs/fold-assoc-funcs.sc

scala> val facLeft  = (accum: Int, x: Int) => accum + x
facLeft: (Int, Int) => Int = <function2>

scala> val facRight = (x: Int, accum: Int) => accum + x
facRight: (Int, Int) => Int = <function2>

scala> val list1 = List(1,2,3,4,5)
list1: List[Int] = List(1, 2, 3, 4, 5)

scala> list1 reduceLeft  facLeft
res0: Int = 15

scala> list1 reduceRight facRight
res1: Int = 15
```

facLeft 和 facRight 函数满足交换律和结合律。两者的区别仅仅在于参数的解释，其函数体相同均为 accum + x。为了清晰，我们定义了这两个函数值并将它们作为参数传递给高阶函数，如 reduceLeft 和 reduceRight。

最后，当对 list1 调用 reduceLeft 并传入 facLeft 作为匿名函数时，我们得到的结果和 facRight 调用 reduceRight 的结果相同。事实上，使用两者中任意一个匿名函数都会得到相同的结果，因为它们满足交换律和结合律。

下面概述一下这几个例子中实际发生的运算。对 list1 调用 reduceLeft 时，我们首先给 facLeft 传入 1 作为累计值 accum（即初始种子值），传入 2 作为 x 的值。匿名函数 facLeft 返回 1 + 2 作为下一次计算时的 accum 值。下一次调用 facLeft 时，传入 3 作为 x 的值，函数返回 3 + 3 或是 6。分析剩下的步骤，可知运算的顺序为：

```
((((1 + 2) + 3) + 4) + 5)   // = 15
```

与此相反，使用 reduceRight 时，5 是初始种子值，我们从右到左依次计算。分析其具体步骤，可以得到运算顺序：

```
((((5 + 4) + 3) + 2) + 1)   // = 15
```

下面重新排列一下表达式，使元素符合其原始顺序。再次将 reduceLeft 的表达式列出，作为对比。注意以下表达式中的括号：

```
((((1 + 2) + 3) + 4) + 5)   // = 15 (reduceLeft的例子)
(1 + (2 + (3 + (4 + 5))))   // = 15 (reduceRight的例子)
```

注意这里的括号是如何反映元素的变量顺序的。稍后我们会回过头来分析这些表达式。

如果采用了一个满足结合律但不满足交换律的函数，会如何呢？

```
scala> val fncLeft  = (accum: Int, x: Int) => accum - x
```

```
fncLeft: (Int, Int) => Int = <function2>

scala> val fncRight = (x: Int, accum: Int) => accum - x
fncRight: (Int, Int) => Int = <function2>

scala> list1 reduceLeft  fncLeft
res0: Int = -13

scala> list1 reduceRight fncRight
res1: Int = -5
```

要了解为什么得到的结果不同，先来分析以上函数中实际发生了什么运算。

如果同样对以上例子列出括号表达式，可以得到如下结果：

```
((((1 - 2) - 3) - 4) - 5)       // = -13 (foldLeft)
((((5 - 4) - 3) - 2) - 1)       // = -5  (foldRight)
(-1 + (-2 + (-3 + (-4 + 5))))   // = -5  (foldRight，重新排列的形式)
```

所以，就像之前的例子，括号清晰地展示了 `reduceLeft` 是从左边开始处理元素的，而 `reduce` 则是从右边处理元素的。

为了表明 `fncLeft` 和 `fncRight` 满足交换律，回想一下 `x - y` 等价于 `x + -y`，`x + -y` 也可以写为 `-y + x`，其结果依然相同：

```
((((1 - 2) - 3) - 4) - 5)      // 原始形式
((((1 + -2) + -3) + -4) + -5)  // 将 x - y 变为 x + -y
(1 + (-2 + (-3 + (-4 + -5))))  // 表明了结合性
```

最后，我们来看看使用既不满足交换律，也不满足结合律的函数时会发生什么。要使用构造 `String` 的函数：

```
scala> val fnacLeft  = (x: String, y: String) => s"($x)-($y)"

scala> val fnacRight = (x: String, y: String) => s"($y)-($x)"

scala> val list2 = list1 map (_.toString)  // 生产String组成的列表

scala> list2 reduceLeft  fnacLeft
res2: String = ((((1)-(2))-(3))-(4))-(5)

scala> list2 reduceRight fnacRight
res3: String = ((((5)-(4))-(3))-(2))-(1)

scala> list2 reduceRight fnacLeft
res4: String = (1)-((2)-((3)-((4)-(5))))
```

再次留意使用 `fnacLeft` 和 `fnacRight` 的不同结果，并确定你理解输出的字符串是如何产生的。

## 尾递归与遍历无限集合

事实证明 `foldLeft` 和 `reduceLeft` 有一个比 `foldRight` 和 `reduceRight` 重大的优势：它们是尾递归的，所以可以从 Scala 的尾递归优化中获益。

为了阐述这一点，回顾一下刚才我们构造的对列表元素做相加的表达式：

```
((((1 + 2) + 3) + 4) + 5)   // = 15 (reduceLeft的例子)
(1 + (2 + (3 + (4 + 5))))   // = 15 (reduceRight的例子)
```

尾递归必须是一次递归中最后一个运行的操作。在 reduceRight 中，最外层的 (1 + …) 在内层嵌套部分完成之前，无法执行，所以这个操作不能被优化为循环，也不是尾递归。在列表中，我们受限于列表的构造方式，只能从头部遍历到尾部。与此相反，reduceLeft 中，我们可以首先对前两个元素执行相加，然后对第三个、第四个做相加，以此类推。换句话说，我们可以将其转为循环，因为这是尾递归。

另一个分析方法是对 Seq 类型实现我们自己的 foldLeft 和 foldRight 方法：

```
// src/main/scala/progscala2/fp/datastructs/fold-impl.sc

// 简化的实现,并未输出集合
// 输入类型是 Seq[A].
def reduceLeft[A,B](s: Seq[A])(f: A => B): Seq[B] = {
  @annotation.tailrec
  def rl(accum: Seq[B], s2: Seq[A]): Seq[B] = s2 match {
    case head +: tail => rl(f(head) +: accum, tail)
    case _ => accum
  }
  rl(Seq.empty[B], s)
}

def reduceRight[A,B](s: Seq[A])(f: A => B): Seq[B] = s match {
  case head +: tail => f(head) +: reduceRight(tail)(f)
  case _ => Seq.empty[B]
}

val list = List(1,2,3,4,5,6)

reduceLeft(list)(i => 2*i)
// => List(12, 10, 8, 6, 4, 2)

reduceRight(list)(i => 2*i)
// => List(2, 4, 6, 8, 10, 12)
```

以上实现并不打算构造 Seq 的子类型。Scala 集合采用我们之前讨论的 CanBuildFrom 技术。否则的话，这里会使用传统的递归模型来实现从左向右和从右向左的遍历。

尽管在实践中，你基本总是会用 Scala 内置的遍历函数，而不是自己写一个。你也应该对这两种遍历及递归有足够的了解，并记住其行为与行为所需的代价。

由于在这里我们处理的是 Seq，所以我们一般只能从左到右进行。Seq.apply(index: Int) 的确可以返回位于索引 index（索引从零开始）的元素，但对于列表来说，每次调用 apply 都需要 $O(N)$ 的复杂度，导致总体的复杂度变为 $O(N^2)$，而不是我们想要的 $O(N)$。所以，foldRight 的实现“推迟”了将当前值与其他递归结果进行运算的时间，直到其他递归完成时才进行。所以，foldRight 不是尾递归的。

对于 foldLeft，我们用嵌套函数 rl 来实现递归。rl 携带一个 accum 参数，用于计算累计

值。当输入不匹配 `head +: tail` 时，就会选择空 `Seq` 的分支，此时返回 `accum` 的值，其中包含了完整的 `Seq[B]`。当我们递归地调用 `rl` 时，这个调用是我们的最后一个操作（尾部调用），因为我们已经先将新元素加到了 `accum` 上，然后再将新的值传入 `rl`。`foldLeft` 是尾递归的。

与此相反，当我们遍历到输入的 `Seq` 的末端时，返回了一个空的 `Seq[B]`，接着随着栈的展开，向该 `Seq[B]` 中插入元素。

最后，值得注意的是，两者输出元素的顺序是不同的。从左到右的递归方法返回了一个将输入列表逆序的输出，这一点也许一开始令你感到惊讶，但这只是因为这个例子将元素插入到序列开头。用 `Vector` 重写以上两个例子，你可以用一样的 `case` 匹配语句。不过不是在开头插入元素，而是在结尾追加元素。这样，新 `foldLeft` 函数输出的 `Vector` 将与输入的 `Vector` 有相同的顺序，而 `foldRight` 的输出则具有相反的元素顺序。

那么，为什么我们要有两种递归方式？如果不必为栈溢出担心，大部分情况下，从右向左的递归可能更适合你的操作。对于序列，`foldRight` 保持了元素的原有顺序。我们可以在调用 `foldLeft` 之后调用 `reverse`，但这意味着遍历两次，而不是一次。这对于大集合来说可能会非常昂贵。

从右向左递归相对从左向右递归还有一个优势。考虑以下情景：你有一个潜在的无限数据流。你不可能将其全部放入内存中的一个集合里，但你可能只需要其中的前 *N* 个元素，而丢弃其他元素。Scala 库的 `Stream` 类型就是为这种目的而设计的。`Stream` 是惰性的，意味着它只在被要求的时候才对其尾部求值（不过对头部的求值是积极的，这一点有时会带来不便）。

这里提到的“求值”是指，在计算时，定义一个无线数据流唯一可能的方法就是使用一个会一直生成数值的函数。该函数可能一直从某个输入渠道，如套接字（就像推特的 firehouse）或大文件中，读取数据或者它本身就是一个能产生数值序列的函数。在下文中我们很快就会看到一个例子。

现在，假设我们定义了一个无限集合。集合中，函数一直生产随机数。Scala 的大部分集合都是严格的或积极的，意味着如果我们在函数中定义一个集合，函数将试图在内存中装下这个集合，从而迅速将内存耗尽。

另一方面，惰性的流只会对集合的头部调用一次随机函数，然后一直等待，直到调用端要求得到集合的尾部值。

现在，我们考虑一个关于无限数字流的有趣例子——斐波那契序列。

斐波那契数的定义如下（这里忽略负数）：

```
f(n) =
  0                 if n = 0
  1                 if n = 1
  f(n-1) + f(n-2)   otherwise
```

像任何良好的递归一样，*n* 等于 0 或 1 时是递归终止的条件，此时 `f(n) = n`，否则 `f(n) = f(n-1) + f(n-2)`。

现在，考虑以下使用 `Stream` 定义函数：

```
// src/main/scala/progscala2/fp/datastructs/fibonacci.sc

scala> import scala.math.BigInt

scala> val fibs: Stream[BigInt] =
     |   BigInt(0) #:: BigInt(1) #:: fibs.zip(fibs.tail).map (n => n._1 + n._2)

scala> fibs take 10 foreach (i => print(s"$i "))
0 1 1 2 3 5 8 13 21 34
```

使用与 cons 操作等价的 `#::`，我们为 `Stream` 构造了序列的前两个元素，分别是 *n* 等于 0 和等于 1 的特例。接着用递归定义了剩下的元素。这个递归是右方向递归。不过我们只取前 *n* 个元素，忽略了其他元素。

在这里，我们定义了 `fibs` 和 `fibs` 的尾部元素：`fibs.zip(fibs.tail).map(…)`。该表达式将 `fibs` 的所有元素和接下来的元素联系在一起。例如：我们有元组如 `(f(2), f(3))`，`(f(3), f(4))` 等，一直到无限，但我们并没有真正对其求值，除非用户要求访问这些值。元组被映射到一个整数，即元组中两个元素之和，这就是 `f(n)` 的下一个值！

最后一行使用 `take` 求出了序列中的前 10 个值（积极的集合也有 `take` 方法）。为了求出 10 个元素，我们必须计算序列尾部 8 次。接着用 `foreach` 对其进行循环，将其打印出来，每个一行。

这个对于递归序列的定义是非巧妙而强大，摘自 `Stream` Scaladoc 页面（http://www.scala-lang.org/api/current/scala/math/Numeric.html）。确保你理解其中的原理，因为这将帮助你理解表达式中各个部分的功能，并能手工计算出前几个值。

注意 fibs 的形式与我们实现的 `foldRight` 很像：`f(0) + f(1) + tail`，这一点很重要。因为这是一个右方向递归，所以我们能在得到想要的元素时停止计算 `tail`。与此不同，试图构造一个惰性求值的左方向递归是不可能的，因为那样会得到这种形式 `f(0 + f(1 +f(tail))`。（对比一下 `foldLeft` 的实现。）所以，右方向递归可以帮助我们处理无限长的、惰性求值的数据流，可以在恰当的时候对数据流进行阶段，而左方向递归则做不到。

> 左方向递归是尾递归，右方向递归则可以提前截断，以处理无限长的、需要惰性求值的数据流。

然而，我们可以注意到有些对序列实现了 `foldRight` 和 `reduceRight` 方法的类型，实际上先进行了逆序，然后再分别执行 `foldLeft` 或 `reduceRight`。例如：`collection.TraversableOnce` 为大部分 `Seq` 提供了这种实现。这将允许我们用尾递归的方式执行右向操作，不过这要花费先执行逆序的开销。

## 6.10　组合器：软件最佳组件抽象

80 年代后期和 90 年代初期，面向对象编程在成为主流后，似乎很有希望在软件复用组

件上有所作为，甚至实现通用的产业组件库。然而，除了少数领域，如不同平台上的 windows API 以外，事情并没有如此发展。

为什么复用组件没有出现？原因很多，但根本原因在于，恰当、通用代码或二进制交互协议是该复用组件的基础，而这种代码或二进制协议并没有出现。事实上，认为对象 API 的丰富性反而破坏了复用组件需要的模块化的观点，是一个悖论。

从更广泛的领域来看，成功的组件模型都依赖于非常简单的基础。数字集成电路（IC）用 $2^n$ 根信号线与信号总线相连，每个信号线是一个布尔值，取值为开或关。在这个极其简单的协议上，人类历史上最具爆炸性发展力的产业诞生了。

HTTP 是组件模型的另一个成功典范。服务间通过狭小但定义良好的接口进行交互，相互传输少数有关数据类型的信息，关于数据本身的格式规定则非常简单。

在这两个例子中。高层协议依赖的基础都很简单，但这个基础使得组件能够创建、生成更复杂的结构。在数字电路中，有些二进制模式被解释为 CPU 指令，有的被解释为内存地址，还有的被解释为数据值。REST（类似 JSON 的数据格式）以及其他高层的标准都基于 HTTP。

对于一座城市，咋看起来，我们认为大部分的建筑都是独一无二的，完全定制化的。事实上，在这些独一无二的建筑背后隐含了众多统一的标准：电力、给排水、房间大小、家具，围绕在这些建筑周围的是标准的街道，在街道上奔驰的拥有自己标准体系的汽车。

面向对象编程从未建立过这种基础性的、通用的标准。在各个语言社区中，组件的基础单位就是对象（有的包含类“模板”，有的没有）。但对象还不够底层。每个开发者都会为 `Customer`（顾客）创造一个新的“标准”。没有任何团队能为 `Customer` 应该有哪些字段达成一致，因为它们需要满足顾客在不同场景下的不同需求，因而需要不同的数据和运算方式。我们需要比对象更底层的东西。

发明跨语言、跨进程的标准组件的努力近年来才出现。类似 CORBA 的模型非常复杂。大部分模型定义了二进制标准，而非代码级的标准。受制于“版本地狱”，这使得交互性变得非常脆弱。问题不在于选择二进制标准而非代码级标准，而在于所定义的二进制标准太过复杂，导致模型的失败。

回想一下本章中所学的示例，其作法恰恰相反。首先我们介绍了一系列集合类型，`List`、`VectorMap` 等。它们共享一些共同的操作，这些操作大多定义在 `Seq` 这个抽象特征中。大部分示例用 `List` 来举例，但实际上选用任意一个集合都可以。

除了 `foreach` 方法，所有操作都是纯净的高阶函数，没有副作用，且都接受函数作为参数，具体工作由该函数完成，如：过滤、转换原始集合中的元素。这种高阶函数与离散数据中的组合器概念非常接近。

我们可以将这些组合器串联起来，从而用很少的代码完成复杂的功能。对于特定问题，我们可以将数据和需要实现的行为分离。这与通常的面向对象编程的方法正好相反，面向对象总是将数据和行为绑定在一起。在自定义的类中创建所需逻辑的临时实现，是面向对象里的典型做法。在本章中，我们已经给出了更好的方法。这就是本章开头引述 Alan J. Perlis 的话的原因。

在本章结束之前，我们来看一个例子，该例是一个简化的工资单计算器：

```
// src/main/scala/progscala2/fp/combinators/payroll.sc

case class Employee (
  name: String,
  title: String,
  annualSalary: Double,
  taxRate: Double,
  insurancePremiumsPerWeek: Double)

val employees = List(
  Employee("Buck Trends", "CEO", 200000, 0.25, 100.0),
  Employee("Cindy Banks", "CFO", 170000, 0.22, 120.0),
  Employee("Joe Coder", "Developer", 130000, 0.20, 120.0))

// 计算每周工资单:
val netPay = employees map { e =>
  val net = (1.0 - e.taxRate) * (e.annualSalary / 52.0) -
    e.insurancePremiumsPerWeek
  (e, net)
}

// “打印”工资单
println("** Paychecks:")
netPay foreach {
  case (e, net) => println(f"  ${e.name+':'}%-16s ${net}%10.2f")
}

// 生成报表:
val report = (netPay foldLeft (0.0, 0.0, 0.0)) {
  case ((totalSalary, totalNet, totalInsurance), (e, net)) =>
    (totalSalary + e.annualSalary/52.0,
      totalNet + net,
      totalInsurance + e.insurancePremiumsPerWeek)
}

println("\n** Report:")
println(f"  Total Salary:    ${report._1}%10.2f")
println(f"  Total Net:       ${report._2}%10.2f")
println(f"  Total Insurance: ${report._3}%10.2f")
```

脚本的输出如下：

```
** Paychecks:
  Buck Trends:        2784.62
  Cindy Banks:        2430.00
  Joe Coder:          1880.00

** Report:
  Total Salary:       9615.38
  Total Net:          7094.62
  Total Insurance:     340.00
```

我们可以用很多方法实现以上逻辑。接下来，我们来考虑一下设计上的几种选择。

首先，尽管本节对面向对象这个组件模型颇多批评，但 OOP 仍然十分有用。我们定义一个 Employee 类来放置雇员所需的字段类型。在真实应用中，这些数据可能要从数据库中加载。

如果我们只用元组来代替自定义的类，会如何呢？你可以试着重写以上代码，并比较其中的差别。命名时使用 Employee 及其字段的名字，可以使代码更具可读性。取有意义的名称是良好悠久的软件设计准则。尽管我强调基础集合类型的优点，但函数式编程并不排斥定义新的类型。与往常一样，设计的取舍应慎重考虑。

然而，Employee 类型很简单，只需要实现少量行为。在经典的面向对象设计中，我们可能会给 Employee 添加很多行为，用于工资计算或实现其他域的逻辑。我相信，我们在这里所选择的设计提供了最佳的分离，同时这种设计也是非常简洁的。如果 Employee 的结构发生变化，需要修改代码时，维护的负担也是非常小的。

还需要注意的是，这段逻辑用一个小脚本即可执行，不需要一个应用程序在多个文件中定义很多类。当然，这个例子仅仅是个玩具。但希望你可以看到，普通的应用并不一定需要大的代码库。

对于使用专用的类型存在一些反对的意见，他们认为这会增加构造实例的开销。在这里，这种开销并不重要。如果我们有十亿条记录，开销还不需要考虑吗？在第 18 章探讨大数据时，我们将再次讨论这个问题。

## 6.11 关于复制

在结束本章之前，让我们考虑一个实际问题。为了保持变量的不变性，对有用的集合进行复制通常是必要的。但假设我有一个包含 100 000 个元素的 Vector，我需要得到一个副本，并替换掉原 Vector 的第 8 个元素。此时我们如果构造一个全新的 100 000 个元素的 Vector 将会是极其低效的。

幸运的是，我们不必付出这个低效的代价，也不必牺牲变量的不可变性。其中的秘诀就是，我们认识到其他 99 999 个元素并没有变化。如果我们能够共享原始 Vector 中的不变部分，而以某种方式表示变化的部分，那么就可以高效地“创建”新 Vector 了。这种思想被称为结构共享。

如果其他线程中的代码正在对原始 Vector 做其他不同的操作，对原始 Vector 的复制不会影响该操作，因为原 Vector 没有被修改。这样，只要对旧版本有一个或多个引用，就可以创建一个 Vector 的“历史”版本。直到对旧版本的引用消失为止，旧版本才会被垃圾回收。

由于历史可以一直保留，使用了结构共享的数据结构被称为持久性数据结构。

我们面临的挑战是选择一种实现数据结构，让我们实现向量语义（或其他数据结构中的语义），同时利用结构共享提供高效的操作方法。让我们来勾勒出底层的数据结构，揭示复制操作是如何工作的。我们不会覆盖所有细节。欲了解更多信息，参见维基百科页面（https://en.wikipedia.org/wiki/Persistent_data_structure）持久性数据结构。

这里采用了分支系数为 32 的树状数据结构。分支因子是每个父节点允许拥有的最大子节

点的数目，意味着搜索和修改操作是 $O(\log_{32}(N))$ 复杂度的，通常这相当于一个常数，甚至对很大的 $N$ 值来说也是如此！

图 6-2 显示了向量（`1,2,3,4,5`）的结构。为了增强阅读性，我们将只使用两个或三个子节点。

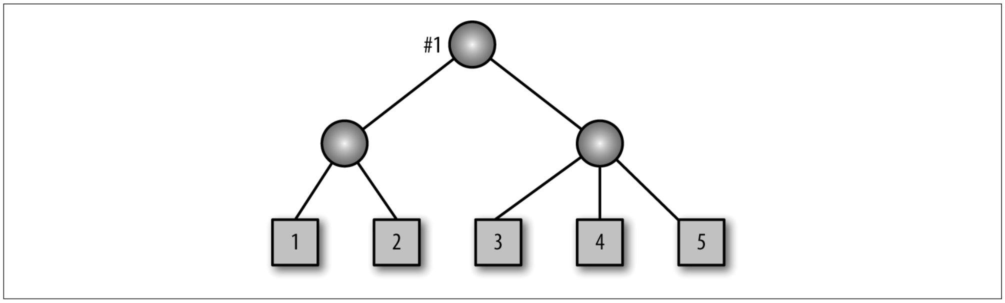



图 6-2：树形结构表示的向量

当你引用向量时，其实你引用的是树的根节点，标记为 `#1`。作为练习，你会进行若干操作，如：通过索引访问特定的元素，执行 `map` 或 `flatMap` 等，这些操作将在树结构的 `Vector` 中实现。

现在假设我们要在 2 和 3 之间插入 2.5。要创建一个新的副本，我们并不需要修改原来的树结构，而是创建新树。图 6-3 显示了当我们在 2 和 3 之间添加 2.5 时发生了什么。

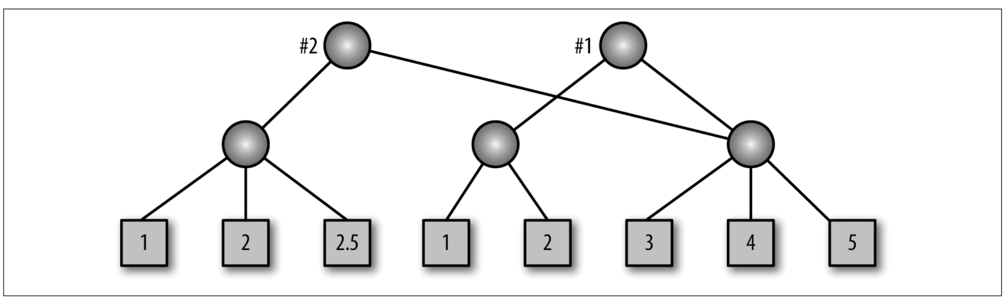



图 6-3：插入新元素前后的 Vector

需要注意的是，原来的树（`#1`）仍然存在，但我们又创建了一个新的根（`#2`）和新的节点。新节点用于连接新根（`#2`）和包含新元素的子节点。这样，我们创建了一个新的左子树，其分支系数为 32，这意味着我们需要在每一层复制多于 32 的子树。但这个复制操作的数量远远小于复制整个树所需的操作。当你需要向一个已有 32 个元素的节点添加元素时，需要做特殊处理。

删除等其他操作与此类似。关于数据结构的好教材都会介绍树操作的标准算法。

因此，如果大规模的不可变数据结构的实现支持高效的复制操作，我们就能够使用它们。相比可变向量，这样的做法会有额外的开销，因为可变向量可以在原地就对元素值进行修改。讽刺的是，这并不意味着，面向对象和过程性的程序必然简单和快速。

由于可变性的危险，在类中用特定的访问方法来包装类的集合是一种常见的做法。但这却增加了代码量和测试的负担。更糟的是，如果集合本身是通过“获取器”方法暴露的，我们在调用时常常会创建保护性副本。保护性副本是通过返回得到的，而不是原集合，这样客户端就无法在对象的控制之外，通过修改集合来改变对象的内部状态。由于集合在非函数式语言的实现中，往往包含效率低下的复制操作，这可能使其程序效率不如对应的函数式程序。

不可变的集合，不仅可以高效地实现，而且它们无需额外的代码来防范有害的修改操作。

其他类型的函数式数据结构，也为高效的复制和适应现代硬件特征做了很多优化，如提高缓存命中率和其他手段。这些被创建的数据结构，有很多被作为可变数据结构的替代者，而可变数据结构总是出现在数据结构和算法的经典教科书上。想要对函数式数据结构有更多了解，请参见 Chris Okasaki 的 *Purely Functional Data Structures* 和 Richard Bird 的 *Pearls of Functional Algorithm Design*（这两本书均由剑桥大学出版社出版），也可以参见 Fethi Rabhi 和 Guy Lapalme 的 *Algorithms: A Functional Programming Approach*（Addison-Wesley 出版社出版）。

## 6.12　本章回顾与下一章提要

我们讨论了函数式编程的基本概念，并指出它们对解决现代软件开发问题的重要性。同时学习了基本集合类型及其高阶函数和组合器如何产生简洁、强大的模块化代码。

典型函数式程序就建立在这个基础之上。在一天结束的时候，所有的程序会输入数据，转换数据，然后输出结果。经典编程语言的许多“仪式”往往掩盖了程序的根本目的。

幸运的是，传统的面向对象的语言已经增加了不少相同的功能。它们中的大多数语言现在已经支持集合中的组合器。Java 8 为 Java 带来了匿名函数（称为 Lambda 表达式），集合类型也增加了高阶函数，从而大大增强了集合的功能。

其次，针对面向对象背景的程序员，我们专门介绍了函数式编程，请参阅 Brian Marick（Leanpub）的《给面向对象程序员的函数式编程介绍》。如果你想说服 Java 开发的朋友来关注函数式编程，你可以考虑阅读我写的 *Introduction to Functional Programming for Java Developers*（O’Reilly 出版）。

Paul Chiusano 和 Rúnar Bjarnason 的 *Functional Programming in Scala*（Manning 出版社出版）一书用 Scala 语言出色、深入地介绍了函数式编程。

练习使用组合器，可以阅读 Phil Gold 的 Ninety-Nine Scala Problems 网页（http://aperiodic.net/phil/scala/s-99/）。

接下来，我们将回到 `for` 表达式，并使用学到的新函数式编程知识，了解 `for` 表达式的实现方式，掌握如何利用它来处理我们自己实现的数据类型，以及 `for` 表达式和组合器产生简洁且强大的代码的奥秘。在这个过程中，我们将深化对函数式编程概念的理解。

我们将在第 16 章再次探究更多函数式编程的高级特性，并在第 12 章深入理解更多关于 Scala 集合的实现细节。

第 7 章

# 深入学习for推导式

我们在 3.6 节中对 `for` 推导式进行了描述。学习完那一节，我们认为它们是很好用且更灵活的 `for` 循环，仅此而已。而实际上这只是冰山的一角，更多复杂的东西被掩藏在水面之下。这些复杂的事物有益于编写出简洁的代码，以优雅的方式解决一些设计上的难题。

在本章中，我们将深入学习被掩藏的知识以便真正理解 `for` 推导式。除了学到 Scala 是如何实现 `for` 推导式之外，我们还将学会如何在自己创建的容器类型中使用它们。

在本章的最后，我们将通过实验揭示到底有多少 Scala 容器类型使用 `for` 推导式解决一些常见的设计难题，例如：如何在执行一组操作时进行错误处理。最后，我们将提炼出一项知名的函数式技巧，该技巧可以用于表示重复习语。

## 7.1　内容回顾：for推导式组成元素

`for` 推导式中包含一个或多个生成器表达式，外加可选的*保护表达式*（guard expression，用于过滤数据）以及值定义。推导式的输出可以用于“生成”新的容器，也可以在每次遍历时执行具有副作用的代码块，如打印输出。下面的例子解释了所有这些特性。该示例移除了文本文件中所有的空行：

```
// src/main/scala/progscala2/forcomps/RemoveBlanks.scala
package progscala2.forcomps

object RemoveBlanks {

  /**
   * 从指定的输入文件中移除空行。
   */
  def apply(path: String, compressWhiteSpace: Boolean = false): Seq[String] =
    for {
```

```
    line <- scala.io.Source.fromFile(path).getLines.toSeq         // ❶
    if line.matches("""^\s*$""") == false                         // ❷
    line2 = if (compressWhiteSpace) line replaceAll ("\\s+", " ")  // ❸
            else line
  } yield line2                                                   // ❹

  /**
   * 从指定的输入文件中移除空行,并将其他行内容依次发送给标准输出。
   * @param 参数列表中包含了文件路径。为每一个文件路径都增加了可选的"-"前缀,
   *            并会压缩以"-"前缀开头文件中的剩余空白符。
   */
  def main(args: Array[String]) = for {
    path2 <- args                                                 // ❺
    (compress, path) = if (path2 startsWith "-") (true, path2.substring(1))
                       else (false, path2)                        // ❻
    line <- apply(path, compress)
  } println(line)                                                 // ❼
}
```

❶ 使用 scala.io.Source 对象（http://www.scala-lang.org/api/current/scala/io/Source$.html）打开文件并读取文件行，getLines 返回 scala.collection.Iterator 对象（http://www.scala-lang.org/api/current/scala/collection/Iterator.html）。由于 for 推导式无法返回 Iterator 对象，for 推导式的返回类型由初始的生成器所决定，因此我们必须将其转化成一个序列。

❷ 使用正则表达式过滤空行。

❸ 定义局部变量。假如未开启空白符压缩，那么局部变量将存储未变的非空行，反之则会将局部变量设置为一个新的行值，该行值已经将所有的空白符压缩为一个空格。

❹ 由于我们使用 yield 方法返回行内容，因此 for 推导式构造了 apply 方法返回的 Seq[String]（http://www.scala-lang.org/api/current/index.html#scala.collection.Seq）。随后我们也将处理 apply 返回的实际容器。

❺ main 方法使用 for 推导式处理参数列表，每个参数都会被视为待处理的文件路径。

❻ 假如文件路径以 - 字符起始，空白符会被压缩，否则只会除去空白行。

❼ 将所有处理后的行内容一起输出到标准输出 stdout。

该文件通过 sbt 被编译。请在 sbt 命令行下运行该文件源代码。首先，尝试运行代码时不加前缀字符 -。我们下面列出了一些输出行的内容：

```
> run-main progscala2.forcomps.RemoveBlanks \
  src/main/scala/progscala2/forcomps/RemoveBlanks.scala
[info] Running ...RemoveBlanks src/.../forcomps/RemoveBlanks.scala
// src/main/scala/progscala2/forcomps/RemoveBlanks.scala
package forcomps
object RemoveBlanks {
  /**
   * 移除指定输入文件的空行。
   */
  def apply(path: String, compressWhiteSpace: Boolean = false): Seq[String] =
...
```

原始文件中的空行已经被移除。运行代码时如果添加前缀 - 将生成下列信息：

```
> run-main progscala2.forcomps.RemoveBlanks \
  src/main/scala/progscala2/forcomps/RemoveBlanks.scala
[info] Running ...RemoveBlanks -src/.../forcomps/RemoveBlanks.scala
// src/main/scala/progscala2/forcomps/RemoveBlanks.scala
package forcomps
object RemoveBlanks {
 /**
 * 从指定的输入文件中移除空白行。
 */
 def apply(path: String, compressWhiteSpace: Boolean = false): Seq[String] =
...
```

现在执行代码后，空白符将会被压缩为一个空格。

你也许希望对该应用进行修改，添加更多的选项，例如，在每一行中增加行号，将输出写到单独的文件，计算统计值等。你该如何将参数数组中各个单独元素转变为类 Unix 风格命令行的选项呢？

我们再看看 apply 方法返回的实际容器。假如运行了 sbt 控制台，我们便能查看该容器：

```
> console
Welcome to Scala version 2.11.2 (Java HotSpot(TM) ...).
...
scala> val lines = forcomps.RemoveBlanks.apply(
     |   "src/main/scala/progscala2/forcomps/RemoveBlanks.scala")
lines: Seq[String] = Stream(
// src/main/scala/progscala2/forcomps/RemoveBlanks.scala, ?)

scala> lines.head
res1: String = // src/main/scala/progscala2/forcomps/RemoveBlanks.scala

scala> lines take 5 foreach println
// src/main/scala/progscala2/forcomps/RemoveBlanks.scala
package forcomps
object RemoveBlanks {
  /**
   * 移除指定输入文件中的空行。
```

apply 方法将返回惰性 Stream 值，这点在 6.9 节中已经介绍了。当 REPL 在打印出 lines 定义信息后打印行内容时，Stream.toString 方法会计算出文件流的头部内容（也就是该文件的注释行），除此之外该方法还会显示一个问号，该问号代表了文件中尚未被计算出的尾部内容。

我们可以要求获取文件的头部内容，之后获取前五行的内容，这也会迫使 Scala 计算出这些行值。由于处理的文件也许会非常庞大，如果将整个文件原封不动地载入内存会消耗大量的内存，因此此处非常适合使用 Stream 类型。不幸的是，万一需要阅读完整的大型内容集，我们就不得不把全部内容都载入内存。这是因为 Stream 会记住它所解析出的所有元素的内容。请注意由于上面出现的两个 for 推导式（分别出现在 apply 和 main 方法中）的每次迭代都不会保存状态，因此并不需要在内容中保存多于一行的数据。

事实上，当你调用 scala.collection.Iterator 的 toSeq 方法时，会调用子类型 scala.collection.TraversableOnce 中的默认实现并返回 Stream 类型对象。而 Iterator 的其他子

类型则可能会返回一个严格型（strict）容器。

## 7.2 for推导式：内部机制

for 推导式的语法实际上是编译器提供的语法糖，它会调用容器方法 foreach、map、flatMap 以及 withFilter 方法。

为什么还需要提供另一种调用这些方法的方式呢？对于那些非平凡序列（nontrival sequence）而言，使用 for 推导式比调用相关 API 编写的代码更加易读易写。

与我们之前见到的 filter 方法一样，withFilter 方法可以对元素进行过滤。假如容器未定义 withFilter 方法，Scala 会使用 filter 方法替代（会出现编译警告）。不过，与 filter 不同，withFilter 方法并不会构造输出容器。为了得到更高的效率，withFilter 会与其他方法一起执行过滤逻辑，这样能够减少一次生成新容器所带来的开销。更具体一点，withFilter 方法会限制允许传递给后续组合器的元素类型域，这些后续组合器包括 map、flatMap、foreach 以及其他的 withFilter 会调用的方法。

为了能了解 for 推导式这颗语法糖里到底封装了些什么东西，我们会先执行一些非正式的比较操作，之后再探讨之前 mapping 中的详细信息。

考虑下面这个简单的 for 推导式：

```
// src/main/scala/progscala2/forcomps/for-foreach.sc

val states = List("Alabama", "Alaska", "Virginia", "Wyoming")

for {
  s <- states
} println(s)
// 结果值:
// Alabama
// Alaska
// Virginia
// Wyoming

states foreach println
// 执行结果与之前一致。
```

注释中列出了输出结果。（从现在开始，我不会再像以前那样频繁地展示 REPL 会话信息。有时我会列出代码，并在注释中展示重要的结果。）

在推导式之后存在一个不含 yield 表达式的生成器表达式，该表达式对应了容器 foreach 方法中执行的表达式。

如果我们使用 yield 操作生成容器，会发生什么呢？

```
// src/main/scala/progscala2/forcomps/for-map.sc

val states = List("Alabama", "Alaska", "Virginia", "Wyoming")

for {
```

```
  s <- states
} yield s.toUpperCase
// 结果值: List(ALABAMA, ALASKA, VIRGINIA, WYOMING)

states map (_.toUpperCase)
// 结果值: List(ALABAMA, ALASKA, VIRGINIA, WYOMING)
```

生成器表达式中包含了 yield 表达式，该生成器对应了一次 map 操作。那么 for 推导式是在什么时候利用 yield 操作构造出新的容器的呢？第一个生成器表达式决定了最终的结果容器类型。通过观察对应的 map 表达式的行为，你会发现这是合理的。如果将输入的 List 类型修改为 Vector 类型，你会发现这将生成一个新的 Vector 容器。

如果我们定义了多个生成器，会发生什么呢？

```
// src/main/scala/progscala2/forcomps/for-flatmap.sc

val states = List("Alabama", "Alaska", "Virginia", "Wyoming")

for {
  s <- states
  c <- s
} yield s"$c-${c.toUpper}"
// 结果值: List("A-A", "l-L", "a-A", "b-B", ...)

states flatMap (_.toSeq map (c => s"$c-${c.toUpper}"))
// 结果值: List("A-A", "l-L", "a-A", "b-B", ...)
```

第二个生成器会遍历字符串 s 中的每一个字符。而设计的 yield 语句将返回各个字符及对应的大写字符，这两个字符通过横线分隔。

如果存在多个生成器，那么除最后一个之外，其他所有的生成器都会被转化成 flatMap 调用。最后一个生成器对应了一次 map 调用。上述代码也将产生 List 对象。你也可以尝试使用其他输入容器类型，看看输出结果是什么类型。

如果我们再添加一个*保护式*（guard），又会发生什么呢？

```
// src/main/scala/progscala2/forcomps/for-guard.sc

val states = List("Alabama", "Alaska", "Virginia", "Wyoming")

for {
  s <- states
  c <- s
  if c.isLower
} yield s"$c-${c.toUpper} "
// 结果值: List("l-L", "a-A", "b-B", ...)

states flatMap (_.toSeq withFilter (_.isLower) map (c => s"$c-${c.toUpper}"))
// 结果值: List("l-L", "a-A", "b-B", ...)
```

请注意，Scala 在最终的 map 调用之前插入了一条 withFilter 调用语句。

最后，如下所示，我们在语句中定义了一个变量：

```
// src/main/scala/progscala2/forcomps/for-variable.sc

val states = List("Alabama", "Alaska", "Virginia", "Wyoming")

for {
  s <- states
  c <- s
  if c.isLower
  c2 = s"$c-${c.toUpper} "
} yield c2
// 结果值: List("l-L", "a-A", "b-B", ...)

states flatMap (_.toSeq withFilter (_.isLower) map { c =>
  val c2 = s"$c-${c.toUpper} "
  c2
})
// 结果值: List("l-L", "a-A", "b-B", ...)
```

## 7.3 for推导式的转化规则

现在我们已经对 for 推导式转化成容器方法的原理有了初步的了解。下面我们将定义更加详细的细节。

首先，在像 pat <- expr 这样的生成器表达式中，pat 实际上是一个*模式表达式*（pattern expression），例如：(x,y) <- List((1,2),(3,4))。Scala 会以类似的方式对值定义语句 pat2 = expr 进行处理，该语句也会被视为某一模式。

Scala 在转化 for 推导式时，要做的第一件事便是将 pat <- expr 语句转化为下列语句：

```
// pat <- expr
pat <- expr.withFilter { case pat => true; case _ => false }
```

之后，Scala 将重复执行下列转化规则，直到所有的推导表达式都被替换掉。值得一提的是，某些转化会生成新的 for 推导式，而后续的迭代则会负责对这些推导式进行转化。

如果 for 推导式中包含了一个生成器和一个 yield 表达式，那么该表达式将被转化为下列语句：

```
// for ( pat <- expr1 ) yield expr2
expr map { case pat => expr2 }
```

如果 for 循环中未使用 yield 语句，但执行的代码具有副作用，那么该语句将被转化为：

```
// for ( pat <- expr1 ) expr2
expr foreach { case pat => expr2 }
```

包含多个生成器（同时包含 yield 表达式）的 for 推导式将被转化成下列语句：

```
// for ( pat1 <- expr1; pat2 <- expr2; ... ) yield exprN
expr1 flatMap { case pat1 => for (pat2 <- expr2 ...) yield exprN }
```

请留意，嵌套的生成器会被转化成嵌套的 for 推导式。这些嵌套的 for 推导式会在下一次执行转化规则时被转化成方法调用。上面示例中 (...) 代表了省略的表达式，这些表达式

可能是其他的生成器，也可能是值定义或保护式（guard）。

包含多个生成器的 for 循环将被翻译成下列语句：

```
// for ( pat1 <- expr1; pat2 <- expr2; ... ) exprN
expr1 foreach { case pat1 => for (pat2 <- expr2 ...) yield exprN }
```

我们之前所见的示例中包含保护式（guard）表达式，该表达式被编写在单独的一行中。事实上，guard 以及上一行中的代码可以编写在一行中，例如：`pat1 <- expr1 if guard`。

后面跟着保护式的生成器会被翻译成下列语句：

```
// pat1 <- expr1 if guard
pat1 <- expr1 withFilter ((arg1, arg2, ...) => guard)
```

此处，变量 `argN` 代表了传递给 `withFilter` 方法的参数。对于大多数的容器而言，传入的方法中只含有一个参数。

如果生成器后面尾随着一个值定义，那么转化这个生成器的复杂度会令人惊奇。如下所示：

```
// pat1 <- expr1; pat2 = expr2
(pat1, pat2) <- for {              // ❶
  x1 @ pat1 <- expr1               // ❷
} yield {
  val x2 @ pat2 = expr2            // ❸
  (x1, x2)                         // ❹
}
```

❶ `for` 推导式将返回包含两个模式的 pair 对象。

❷ `x1 @ pat1` 语句会将整个表达式中 `pat1` 所匹配的值赋给变量 `x1`，该值可能包含另一个变量的某一部分。假如 `pat1` 是一个不可变变量名，`x1` 和 `pat1` 的赋值将会是冗余的。

❸ 将 `pat2` 值赋给 `x2`。

❹ 返回元组。

下面的 REPL 会话中包含了 `x @ pat = expr` 语句的对应示例：

```
scala> val z @ (x, y) = (1 -> 2)
z: (Int, Int) = (1,2)
x: Int = 1
y: Int = 2
```

变量 `z` 的值为元组 `(1,2)`，而变量 `x` 和变量 `y` 则对应了元组中各个组成部分的值。

由于很难讲清楚 `for` 推导式完整的转化过程，所以我们从一个具体的例子开始学起。

```
// src/main/scala/progscala2/forcomps/for-variable-translated.sc

val map = Map("one" -> 1, "two" -> 2)

val list1 = for {
  (key, value) <- map   // 本行和下一行将会被翻译成什么语句呢？
  i10 = value + 10
} yield (i10)
// 执行结果值: list1: scala.collection.immutable.Iterable[Int] = List(11, 12)
```

```
// 翻译后的语句:
val list2 = for {
  (i, i10) <- for {
    x1 @ (key, value) <- map
  } yield {
    val x2 @ i10 = value + 10
    (x1, x2)
  }
} yield (i10)
// 执行结果: list2: scala.collection.immutable.Iterable[Int] = List(11, 12)
```

请留意外围的 `for {...}` 中的两个表达式的转化方式。尽管在内部表达式中我们返回了 `(x1, x2)` 对，但事实上只返回了 `x2` 变量（等价于 `i10` 变量）。另外，我们知道 `Map` 对象包含一组键值对元素，因此在上面的代码中，与生成器所匹配的模式类型为键值对类型，我们也使用该类型遍历 map 对象。

这便是 `for` 推导式完整的转化规则。你可以应用这些规则，将 `for` 推导式转化成针对容器的一组方法调用。你不必经常执行这样的转化，不过有时候这样做能帮助你调试问题。

我们再看一个示例，该示例应用模式匹配对一个常见的格式为 key = value 的属性文件进行解析。

```
// src/main/scala/progscala2/forcomps/for-patterns.sc

val ignoreRegex = """^\s*(#.*|\s*)$""".r                          // ❶
val kvRegex = """^\s*([^=]+)\s*=\s*([^#]+)\s*.*$""".r             // ❷

val properties = """
  |# Book properties
  |
  |book.name = Programming Scala, Second Edition # A comment
  |book.authors = Dean Wampler and Alex Payne
  |book.publisher = O'Reilly
  |book.publication-year = 2014
  |""".stripMargin                                                 // ❸

val kvPairs = for {
  prop <- properties.split("\n")                                  // ❹
  if ignoreRegex.findFirstIn(prop) == None                        // ❺
  kvRegex(key, value) = prop                                      // ❻
} yield (key.trim, value.trim)                                    // ❼
// Returns: kvPairs: Array[(String, String)] = Array(
//   (book.name,Programming Scala, Second Edition),
//   (book.authors,Dean Wampler and Alex Payne),
//   (book.publisher,O'Reilly),
//   (book.publication-year,2014))
```

❶ 正则表达式用于查找将被“忽略”掉的行，例如：空行或注释行。表达式中的 `#` 是注释符，只有当它在该行所有非空白符中位于第一位时才能匹配该表达式。

❷ 用于匹配 `key = value` 对的正则表达式，该表达式可以处理包含任意多的空白字符以及注释的情况。

❸ 一个输入示例，该示例中包含了多行属性字符串。请留意，为了能够移除 | 字符以及该字符之前的所有的行首字符，代码中使用 `StringLike.stripMargin`（http://www.scala-lang.org/api/current/scala/collection/immutable/StringLike.html）方法。运用这项技术，我们可以将各行缩进对齐，而且无需担心这些空白字符会作为字符串的一部分被解析。

❹ 各个属性通过换行符分隔。

❺ 过滤各行字符串，只留下我们不希望被忽略掉的行。

❻ 本行左侧代码运用了模式表达式；通过正则表达式从有效的属性行中抽取出键及值对应的字符串。

❼ 生成最终的键值对并删掉剩余无用的空白字符。

由于生成器调用了返回 `Array` 对象的 `String.split` 方法，因此上述示例返回了 `Array[(String, String)]` 类型的对象。

请查看 Scala 语言规范（http://www.scala-lang.org/docu/files/ScalaReference.pdf ）的 6.19 节，该节中列举了更多关于 `for` 推导式及其转化原理的相关示例。

## 7.4 Option以及其他的一些容器类型

我们在示例中使用了 `List`、`Array` 以及 `Map` 容器，不过除了这些明显的容器类型之外，`for` 推导式中还可以使用任何一种实现 `foreach`、`map`、`flat` 以及 `withFilter` 方法的类型。换言之，任何提供了这些方法的类型均可视为容器，而我们也可以在 `for` 推导式中使用这些类型的实例。

我们将学习一些其他类型的容器。了解对这些容器应用 `for` 推导式会对代码造成多么难以执行的改变。

### 7.4.1 Option容器

`Option` 是一个二元容器，其中也许包含了一个元素，也许不包含任何元素。`Option` 提供了我们所需的四个方法。

下面列出了 `Option` 类型中所需方法的实现代码（代码摘自 Scala 2.11 版本库源代码，一些无关的细节已省略或修改）：

```
sealed abstract class Option[+A] { self =>                    // ❶
  ...
  def isEmpty: Boolean  // Some和None类型会实现该变量。

  final def foreach[U](f: A => U): Unit =
    if (!isEmpty) f(this.get)

  final def map[B](f: A => B): Option[B] =
    if (isEmpty) None else Some(f(this.get))

  final def flatMap[B](f: A => Option[B]): Option[B] =
    if (isEmpty) None else f(this.get)
```

```
  final def filter(p: A => Boolean): Option[A] =
    if (isEmpty || p(this.get)) this else None

  final def withFilter(p: A => Boolean): WithFilter = new WithFilter(p)

  /** 为了能够遵守“不创建新容器”的约定,我们需要声明WithFilter类。
   *  尽管Option容器的最大元素数为1,创建新容器似乎也不会对性能造成多大影响。
   */
  class WithFilter(p: A => Boolean) {
    def map[B](f: A => B): Option[B] = self filter p map f      // ❷
    def flatMap[B](f: A => Option[B]): Option[B] = self filter p flatMap f
    def foreach[U](f: A => U): Unit = self filter p foreach f
    def withFilter(q: A => Boolean): WithFilter =
      new WithFilter(x => p(x) && q(x))
  }
}
```

❶ self => 表达式定义了 Option 实例的一个别名，该别名在后面出现的 WithFilter 方法中被使用。如果想了解更多信息，请查看 14.6 节。

❷ 我们需要在封闭的 Option 实例中使用之前定义的 self 引用，而不是在 WithFilter 实例中使用。也就是说，如果我们使用 this 引用，该引用将指向 WithFilter 实例。

final 关键字会阻止子类覆写这些方法实现。当你看到 Option 这个基类中引用了继承类时，也许会感到些许震惊。因为通常情况下，如果基类知道继承类型的所有信息，该设计会被视为不好的面向对象设计。

不过，我们可以回顾下第 2 章关于 sealed 关键字的内容。sealed 关键字意味着 Scala 只允许在相同文件中定义该类的子类。Option 对象要么是空对象（None），要么是非空对象（Some）。因此，这段代码是健壮、全面（能覆盖所有的场景）、简洁而且完全合理的。

这些 Option 方法具有一个重要的特性：只有当 Option 非空时，那些方法才会使用传入的函数参数。

利用这一特性，我们能够优雅地解决一个常见的设计问题。分布式计算领域中有一个常见的模式，即将计算分解为小任务，再将这些任务分发到集群中，之后再收集这些任务的执行结果。例如：Hadoop 的 MapReduce 框架（http://hadoop.apache.org）就使用了这一模式。我们希望能通过一种优雅的方式忽略任务结果为空的情况，只对非空结果进行处理。暂且会忽略那些出错的任务。

首先，假设每个任务都会返回 Option 对象，其中 None 对象代表了结果为空的返回值，而 Some 对象则对非空结果进行了封装。之后，我们希望以最优雅的方式过滤出非空结果。

在下面的示例中，有一个包含了三个结果值的集合，其中每个结果值均为 Option[Int] 对象：

```
// src/main/scala/progscala2/forcomps/for-options-seq.sc

val results: Seq[Option[Int]] = Vector(Some(10), None, Some(20))

val results2 = for {
  Some(i) <- results
```

```
} yield (2 * i)
// 执行结果: Seq[Int] = Vector(20, 40)
```

`Some(i) <- list` 语句会对 `results` 变量中包含的元素执行模式匹配，移除 `None` 元素，并抽取类型为 `Some` 的元素的整数值。之后，生成我们所希望得到的最终表达式。而该程序输出为 `Vector(20, 40)`。

下面我们做一个练习，回顾一遍 `for` 推导式的转化规则。首先，我们运用第一条规则，将每一个格式为 `pat <- expr` 的表达式转化成一个包含 `withFilter` 语句的表达式：

```
// Translation step #1
val results2b = for {
  Some(i) <- results withFilter {
    case Some(i) => true
    case None => false
  }
} yield (2 * i)
// 执行结果: results2b: List[Int] = List(20, 40)
```

最后，我们将格式为 `for { x <- y } yield (z)` 的表达式转化成一个 `map` 调用。

```
// Translation step #2
val results2c = results withFilter {
  case Some(i) => true
  case None => false
} map {
  case Some(i) => (2 * i)
}
// 执行结果: results2c: List[Int] = List(20, 40)
```

实际上，这条 `map` 表达式会生成一条编译器警告信息：

```
<console>:9: warning: match may not be exhaustive.
It would fail on the following input: None
       } map {
             ^
```

如果传递给 `map` 方法的*偏函数*（partial function）未使用 `None => ...` 子句，这种情况通常会比较危险。但如果 `map` 方法处理的元素出现了 `None` 对象，Scala 又会抛出 `MatchError`（http://www.scala-lang.org/api/current/scala/MatchError.html）的异常。不过，由于调用 `withFilter` 的方法中已经移掉了所有的 `None` 元素，运行代码时便不会出现这一错误。

现在让我们再思考另一个设计难题。这个难题并不是关于忽略各个独立任务中毫无关联的空值并组合非空值的问题。问题是在执行一组非独立的操作步骤中，我们希望在获得了一个 `None` 值时，能够尽快停止整个相互关联的处理过程。

`None` 对象存在一个局限，就是你无法知道为什么这一操作不返回任何值。原因可能是出错导致了返回 `None` 值。针对这一局限，我们会在本章的后续内容中解决。

我们也可以编写复杂的条件逻辑代码，每次处理一个输出并检查结果值[1]。不过使用一个 `for` 推导式是更好的做法。

---

注 1：请查看代码示例文件 src/main/scala/progscala2/forcomps/for-options-bad.sc>。

```
// src/main/scala/progscala2/forcomps/for-options-good.sc

def positive(i: Int): Option[Int] =
  if (i > 0) Some(i) else None

for {
  i1 <- positive(5)
  i2 <- positive(10 * i1)
  i3 <- positive(25 * i2)
  i4 <- positive(2  * i3)
} yield (i1 + i2 + i3 + i4)
// 执行结果: Option[Int] = Some(3805)

for {
  i1 <- positive(5)
  i2 <- positive(-1 * i1)              // ❶   失败!
  i3 <- positive(25 * i2)              // ❷
  i4 <- positive(-2 * i3)              // 失败!
} yield (i1 + i2 + i3 + i4)
// 执行结果: Option[Int] = None
```

❶ 将返回 `None` 值，“左箭头”执行了什么操作呢？

❷ 该行代码中引用了 `i2` 变量，这合理吗？

`positive` 函数返回一个 `Option[Int]` 对象。同时，假如输入值 `i` 是正数，`positive` 也返回 `Some(i)` 对象，否则将返回 `None` 对象。

请留意这两个 `for` 推导式中第二个和第三个表达式。这两个表达式使用了之前表达式的结果值。这些表达式似乎都认为程序将按照“正常流程”运行，因此使用从 `Option[Int]` 对象中抽取出的 `Int` 值是安全的。

我们认为第一个 `for` 推导式能正常执行。而第二个 `for` 推导式也能正常执行！一旦返回了 `None` 值，后续的表达式将会停止运行。这是因为 `map` 或 `flatMap` 不会对这些函数字面量进行处理。

接下来我们将学习其他三个具有相同属性的容器类型：`Either`（http://www.scala-lang.org/api/current/scala/util/Either.html）、`Try`（http://www.scala-lang.org/api/current/scala/util/Try.html）以及 `Validation`（http://docs.typelevel.org/api/scalaz/stable/7.0.4/doc/#scalaz.Validation）类型。名为 Scalaz 的第三方库会对这三个类型进行定义，且该库很受欢迎。

### 7.4.2 Either：Option类型的逻辑扩展

我们注意到 `Option` 类型有一个弊端，即 `None` 对象不能提供任何信息告诉我们为什么不返回值。例如：由于发生错误，返回 `None` 对象的情况。使用 `Either` 替代 `Option` 是一种解决方案。与 Either 的英文字面意思一样，`Either` 是一类能且只能持有两种事物中一种的容器。换言之，`Option` 能持有 0 个或 1 个元素，而 `Either` 则持有这个或那个元素项。

`Either` 是一个包含两个参数的参数化类型，该类型的签名为 `Either[+A, +B]`，其中 `A` 和 `B` 是 `Either` 对象中可能持有元素的类型。我们回顾一下，`+A` 表示 `Either` 是类型参数 `A` 的协变（covariant），`+B` 亦是如此。这意味着如果你需要类型 `Either[Any,Any]` 的值，你可以通

过使用类型 Either[String,Int] 得到。这是因为 String 和 Int 类型都是 Any 类型的子类型。所以 Either[String,Int] 是 Either[Any,Any] 的子类型。

Either 同时也是 sealed 抽象类（只能在相同文件中声明子类）。Either 有两个子类 Left[A]（http://www.scala-lang.org/api/current/scala/util/Left.html） 和 Right[B]（http://www.scala-lang.org/api/current/scala/util/Right.html）。我们通过这两个子类从两个可能的元素中选择一种。

Either 概念的产生时间早于 Scala。很长时间以来它被认为是抛出异常的一种替代方案。为了尊重历史习惯，当 Either 用于表示错误标志或某一对象值时，Left 值用于表示错误标志，如：信息字符串或下层库抛出的异常；而正常返回时则使用 Right 对象。很明显，Either 可以用于任何需要持有某一个或另一个对象的场景中，而这两个对象的类型可能不同。

在我们深入了解 Either 类型的某些特殊点之前，我们先用 Either 类型把之前的示例重写一遍。首先，假如你持有一组 Either 对象并希望忽略掉错误值（Left 对象），一个简单的 for 推导式便能做到这点。

```
// src/main/scala/progscala2/forcomps/for-eithers-good.sc

def positive(i: Int): Either[String,Int] =
  if (i > 0) Right(i) else Left(s"nonpositive number $i")

for {
  i1 <- positive(5).right
  i2 <- positive(10 * i1).right
  i3 <- positive(25 * i2).right
  i4 <- positive(2  * i3).right
} yield (i1 + i2 + i3 + i4)
// 执行结果: scala.util.Either[String,Int] = Right(3805)

for {
  i1 <- positive(5).right
  i2 <- positive(-1 * i1).right   // 失败!
  i3 <- positive(25 * i2).right
  i4 <- positive(-2 * i3).right   // 失败!
} yield (i1 + i2 + i3 + i4)
// 执行结果: scala.util.Either[String,Int] = Left(nonpositive number -5)
```

除了对类型进行了适当的修改之外，这一版本的实现与之前使用 Option 的实现非常相似。与 Option 实现类似，我们只能看到第一个错误。不过，注意我们需要调用 postive 方法返回值的 right 方法。要理解这样做的原因，我们需要先了解 right 方法和与之对应的 left 方法的作用。

下面列举了一些关于 Either、Left 和 Right 对象的简单示例，这些示例均取自于 Scaladoc：

```
scala> val l: Either[String, Int] = Left("boo")
l: Either[String,Int] = Left(boo)

scala> val r: Either[String, Int] = Right(12)
r: Either[String,Int] = Right(12)
```

我们声明了两个 Either[String， Int] 对象，其中一个对象赋予了 Left[String] 值，另一

个赋予了 Right[Int] 值。

顺便提一下，之前在 4.6.1 节中我们曾经提到过：如果一个类型中包含两个参数，那么它可以使用中缀表示法表示类型说明。因此，我们可以用下列两种方式声明 l：

```
scala> val l1: Either[String, Int] = Left("boo")
l1: Either[String,Int] = Left(boo)

scala> val l2: String Either Int = Left("boohoo")
l2: Either[String,Int] = Left(boohoo)
```

为了更好的表示类型，我希望可以将 Either 类型更名为 Or 类型！假如你也青睐于 Or，你可以在代码中使用类型别名：

```
scala> type Or[A,B] = Either[A,B]
defined type alias Or

scala> val l3: String Or Int = Left("better?")
l3: Or[String,Int] = Left(better?)
```

Either 本身并未定义组合方法 map、fold 等，我们只能访问 Either.left 或 Either.right 中的组合方法。之所以如此，是因为我们的组合方法只接受单个函数参数，但我们需要为 Either 容器指定两个函数参数。当 Either 值为 Left 值时调用一个，而 Either 值为 Right 值时调用另外一个。Either 对象提供了 left 和 right 方法，这两个方法会构建出一个提供组合方法的*投影对象*（projection）：

```
scala> l.left
res0: scala.util.Either.LeftProjection[String,Int] = \
  LeftProjection(Left(boo))

scala> l.right
res1: scala.util.Either.RightProjection[String,Int] = \
  RightProjection(Left(boo))

scala> r.left
res2: scala.util.Either.LeftProjection[String,Int] = \
  LeftProjection(Right(12))

scala> r.right
res3: scala.util.Either.RightProjection[String,Int] = \
  RightProjection(Right(12))
```

需要注意的是，Either.LeftProjection 值（http://www.scala-lang.org/api/current/index.html#scala.util.Either$）既可以持有 Left 实例，也可以持有 Right 实例。Either.RightProjection 对象（http://www.scala-lang.org/api/current/index.html#scala.util.Either$）与 Either.LeftProjection 相同。下面我们将调用这些投影对象的 map 方法用于传入一个函数参数：

```
scala> l.left.map(_.size)
res4: Either[Int,Int] = Left(3)

scala> r.left.map(_.size)
res5: Either[Int,Int] = Right(12)
```

```
scala> l.right.map(_.toDouble)
res6: Either[String,Double] = Left(boo)

scala> r.right.map(_.toDouble)
res7: Either[String,Double] = Right(12.0)
```

如果调用 LeftProjection.map 方法时，Either 对象持有 Left 实例，map 方法会作用于 Left 实例所持有的对象。这与 Option.map 方法处理 Some 对象的方式类似。不过，如果你调用 LeftProjection 对象的 map 方法，但 Either 对象却持有 Right 实例时，Either 对象会向 map 方法传递 Right 实例持有的对象。这与 Option.map 方法处理 None 对象的方式是类似的。

与之相似，如果调用 RightProjection.map 方法时，Either 对象持有 Right 实例，map 方法会作用于 Right 实例所持有的值。而如果 Either 对象持有 Left 实例时，则依旧如此。

请留意这些操作的返回类型。由于 l.left.map(_.size) 方法将 String 对象转化成了整型（Int）对象，新生成的 Either 对象类型变成 Either[Int,Int]。由于该函数不会对 Right[Int] 对象进行操作，因此第二个类型参数保持不变。

与之类似，由于 r.right.map(_.toDouble) 操作会将 Int 对象转化为 Double 对象，Either[String,Double] 类型会被返回。Scala 提供了 String.toDouble 方法，来对字符串进行解析并返回双精度浮点数。假如解析失败，该方法便会抛出异常。不过，本书后面的章节中不会应用此方法。

我们也可以使用 for 推导式计算出字符串的长度。下面我们列举了之前的表达式以及与其等价的 for 推导式：

```
l.left map (_.size)            // Returns: Left(3)
for (s <- l.left) yield s.size // Returns: Left(3)
```

### 抛出异常还是返回Either值

Either 类型有其可取之处，不过如果代码出错的话，直接抛出异常不是更简单吗？在某些时候，抛出异常当然是更合理的。抛出异常能避免对错误数据进行计算；而有时候调用栈中的某些对象捕获异常可以对故障执行合理的恢复。

不过，抛出异常会破坏引用的透明性。我们一起来看看下面这个精心设计的示例：

```
// src/main/scala/progscala2/forcomps/ref-transparency.sc

scala> def addInts(s1: String, s2: String): Int =
     |   s1.toInt + s2.toInt
addInts: (s1: String, s2: String)Int

scala> for {
     |   i <- 1 to 3
     |   j <- 1 to i
     | } println(s"$i+$j = ${addInts(i.toString,j.toString)}")
1+1 = 2
2+1 = 3
2+2 = 4
3+1 = 4
3+2 = 5
```

```
3+3 = 6

scala> addInts("0", "x")
java.lang.NumberFormatException: For input string: "x"
...
```

看上去我们似乎不需要调用 addInts 函数，只在该函数的调用处直接使用函数值即可。只是我们无法缓存之前的调用并返回命中的缓存值。但是，假如我们向 addInts 方法传递的字符串参数无法解析成 Int 值时，addInts 方法会抛出异常。因此，我们不能使用函数值替代函数调用。对于某些参数列表，这些函数值无法返回。

更糟的是，我们无法从 addInts 方法的类型签名处获知该函数可能会产生的问题。尽管这是一个精心设计的示例，但是终端用户输入的字符串必然会导致函数出错。

的确，Java 提供的*可检查异常*（checked exception）能解决这一个特定问题。Java 的方法签名能通过抛出异常暗示可能会产生错误的条件。不过由于种种原因，可检查异常实际上未能得到很好的使用。而包括 Scala 在内的其他语言也未提供这一功能。Java 程序员也常常避免使用这一功能，改而抛出不可检查的 java.lang.RuntimeException（http://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html）类的子类。

使用 Either 对象，我们可以保障引用透明性，并通过类型签名提醒调用者可能会出现错误。在下面的代码中，我们使用 Either 对象重写了 addInts 方法：

```
// src/main/scala/progscala2/forcomps/ref-transparency.sc

scala> def addInts2(s1: String, s2: String): Either[NumberFormatException,Int]=
     |   try {
     |     Right(s1.toInt + s2.toInt)
     |   } catch {
     |     case nfe: NumberFormatException => Left(nfe)
     |   }
addInts2: (s1: String, s2: String)Either[NumberFormatException,Int]

scala> println(addInts2("1", "2"))
Right(3)

scala> println(addInts2("1", "x"))
Left(java.lang.NumberFormatException: For input string: "x")

scala> println(addInts2("x", "2"))
Left(java.lang.NumberFormatException: For input string: "x")
```

现在，addInts2 方法的类型签名能够提示可能会出现错误。它不再通过抛出异常来捕获调用堆栈中某些应用的控制权，而是将异常作为调用堆栈中的结果值返回，以此来消除程序错误。

现在，我们不仅能够处理正确的字符串输入，使用结果值替代方法调用。我们甚至可以处理无效的字符串输入，使用合适的 Left[java.lang.NumberFormatException] 值代替方法调用！

因此，假如发生了某种错误，我们可以使用 Either 类型来维护调用堆栈的控制权。同时，使用 Either 类也能使客户更清楚地理解 API 的行为。

我们再阅读一遍 addInts2 的实现。在 Java 和 Scala 类库中，抛出异常是一种很常见的做法，因此我们也会编写出 try{...} catch{...} 这样的样板代码，并将好的和不好的结果都封装在 Either 对象中。为了处理异常，我们也许应该使用某些类型将这些样板代码进行封装，而无论操作成功还是失败，这些类型名能够更清楚地表达这当前的状况。Try 类型就做到了这点。

## 7.4.3 Try类型

scala.util.Try（http://www.scala-lang.org/api/current/scala/util/Try.html）的结构与 Either 相似，Try 是一个 sealed 抽象类。它有两个子类，分别为 Success（http://www.scala-lang.org/api/current/scala/util/Success.html）类和 Failure（http://www.scala-lang.org/api/current/scala/util/Failure.html）类。

Success 类的使用方式与 Right 类相似，它会保存正常的返回值。Failure 则与 Left 类相似，不过 Failure 总是保存 Throwable 类型的值。

下面列出了这些类型的签名信息（省略了与本章节内容无关的 trait 定义）。

```
sealed abstract class Try[+T] extends AnyRef {...}
final case class Success[+T](value: T) extends Try[T] {...}
final case class Failure[+T](exception: Throwable) extends Try[T] {...}
```

请注意，不同于 Either[+A, +B] 类，上面这些类中只包含了一种类型参数，这是因为与 Left 类型相对应的类型是 Throwable 类型。

此外，不同于 Either 类，Try 很明显是非对称的类型。该类型只包含了一个“正常”类型（T）以及用于错误场景的 java.lang.Throwable（http://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html）类型。这意味着如果 Try 是一个 Success 类型，Try 就可以定义类似 map 方法这样的组合方法。

和之前一样，我们将使用 Try 重写之前的示例，以学习如何使用 Try。首先，假如你有一组 Try 值，你希望能够忽略其中的 Failure 对象，那么使用一个简单的 for 推导式就能做到这点：

```
// src/main/scala/progscala2/forcomps/for-tries-good.sc
import scala.util.{ Try, Success, Failure }

def positive(i: Int): Try[Int] = Try {
  assert (i > 0, s"nonpositive number $i")
  i
}

for {
  i1 <- positive(5)
  i2 <- positive(10 * i1)
  i3 <- positive(25 * i2)
  i4 <- positive(2  * i3)
} yield (i1 + i2 + i3 + i4)
// 返回值: scala.util.Try[Int] = Success(3805)
```

```
for {
  i1 <- positive(5)
  i2 <- positive(-1 * i1)             // 失败!
  i3 <- positive(25 * i2)
  i4 <- positive(-2 * i3)             // 失败!
} yield (i1 + i2 + i3 + i4)
// 返回值: scala.util.Try[Int] = Failure(
//   java.lang.AssertionError: assertion failed: nonpositive number -5)
```

请留意 positive 方法的具体定义。假如断言失败，Try 代码块会返回 Failure 对象，该对象封装了可抛出的 java.lang.AssertionError 对象（http://docs.oracle.com/javase/8/docs/api/java/lang/AssertionError.html）。否则，Try 表达式的结果值将被封装在 Success 中。下面列举了 positive 方法的另一个实现版本，该实现方法更明确地表明了 Try 类型的处理逻辑：

```
def positive(i: Int): Try[Int] =
  if (i > 0) Success(i)
  else Failure(new AssertionError("assertion failed"))
```

for 推导式看上去和之前 Option 示例中的推导式完全一致。通过类型推导，代码也变得非常精简。你也可以集中精力对“正确逻辑”进行处理，让 Try 负责捕获错误。

### 7.4.4　Scalaz提供的Validation类

存在这样一种场景：我们之前讨论的所有类型都无法很好地满足我们的需求。假如 for 推导式中出现了空值（由 Option 类型提供）或错误，那么组合器（combinator）便不会调用后续的表达式。事实上，我们会在发生第一个错误时便停止执行后续代码。不过，假如我们正在执行一些相互独立的操作，并希望执行这些操作时收集所有发生的错误，待操作执行完成后再决定如何处理错误，那该怎么办呢？对用户输入进行验证便是一个典型的应用场景，这些用户输入可能源自网页表单。这样，你可以一次将所有的错误信息返回给用户。

Scala 标准库并未提供能满足这类场景的类型，不过 Scalaz（http://github.com/scalaz/scalaz）这一广受欢迎的第三方库提供了能满足这一需求的 Validation（http://docs.typelevel.org/api/scalaz/stable/7.0.4/doc/#scalaz.Validation）类型。

```
// src/main/scala/progscala2/forcomps/for-validations-good.sc
import scalaz._, std.AllInstances._

def positive(i: Int): Validation[List[String], Int] = {
  if (i > 0) Success(i)                                     // ❶
  else Failure(List(s"Nonpositive integer $i"))
}

for {
  i1 <- positive(5)
  i2 <- positive(10 * i1)
  i3 <- positive(25 * i2)
  i4 <- positive(2  * i3)
} yield (i1 + i2 + i3 + i4)
// 返回值: scalaz.Validation[List[String],Int] = Success(3805)

for {
```

```
  i1 <- positive(5)
  i2 <- positive(-1 * i1)             // 错误!
  i3 <- positive(25 * i2)
  i4 <- positive(-2 * i3)             // 错误!
} yield (i1 + i2 + i3 + i4)
// 返回: scalaz.Validation[List[String],Int] =
//   Failure(List(Nonpositive integer -5))                    // ❷

positive(5) +++ positive(10) +++ positive(25)                 // ❸
// 返回: scalaz.Validation[String,Int] = Success(40)

positive(5) +++ positive(-10) +++ positive(25) +++ positive(-30)    // ❹
// 返回: scalaz.Validation[String,Int] =
//   Failure(Nonpositive integer -10, Nonpositive integer -30)
```

❶ Success 和 Failure 都是 scalaz.Validation（http://docs.typelevel.org/api/scalaz/stable/7.0.4/doc/index.html#scalaz.Validation）的子类，并不是 scala.util.Try（http://www.scala-lang.org/api/current/scala/util/Try.html）的子类。

❷ 因为我们运用了 for 推导式，所以一旦发生错误，计算过程还是会立刻停止。因此我们无法看到最后那条关于 i4 变量的错误。

❸ 不过，在该表达式以及后续的表达式中，我们调用了 positive 方法的计算值，并将结果累加。如果存在错误，则将错误叠加起来。

❹ 结果值中同时包含了表达式中存在的两个错误。

与 Either 类型相似，Validation 类中的第一个参数表示用于汇报错误的类型。在这个示例中，由于我们使用了 List[String] 类型作为类型参数，因此我们能够将多个错误叠加起来。不过，使用 String 或其他任何支持追加操作的集合来作为类型参数同样可以达到叠加错误的效果。Scalaz 则负责调用合适的“连接”方法。

第二个类型参数代表了验证通过后的返回值类型，在这个示例中我们使用了 Int 类型，不过我们也可以使用集合类型。

请注意，如果表达式存在异常，for 推导式会立刻完成对该语句的估值。不过由于每次调用 positive 方法时都需要前一次的执行结果，因此我们仍然能够得到期望的结果。

不过，我们之后会看到如何使用 +++“加法”操作符[2]执行各个独立估值。假如输入信息能通过所有的验证，我们将基于这些结果值统计出最终结果。反之，该表达式便会将所有的错误归纳起来，并将叠加后的结果作为表达式的结果值。为此，我们使用了一组字符串存储所有的错误信息。

你无法在网页表单中计算出这些数字的总和，只能将这些字段值汇聚在一起。我们将对这个示例进行修改，使其更加符合表单验证的真实场景。List[(String,Any)] 类型作为验证通过后返回的类型，是一组键值元组列表。假如验证成功，我们便可以调用 List 类型提供的 toMap 方法生成一个 Map 对象，并将新生成的对象返还给调用者。[3]

---

注 2：这只是 Scalaz 中多个可选技术中的一个，请查阅 Validation Scaladoc（http://docs.typelevel.org/api/scalaz/stable/7.0.4/doc/index.html#scalaz.Validation）文档学习其他的相关示例。

注 3：为什么不使用 Map[String, Any] 类型呢？看来 Scalaz 并不支持这一选择。

我们将对用户的姓名和年龄进行验证。用户的姓氏及名字均不能为空，并且只能包含字母。而用户的年龄可能是从网页表单中抽取出来的，它的起始类型是字符串类型，该字符串必须能够被解析成一个正整数：

```
// src/main/scala/progscala2/forcomps/for-validations-good-form.sc
import scalaz._, std.AllInstances._

/** 对用户名进行验证;用户名必须非空并且只能包含字母。*/
def validName(key: String, name: String):
    Validation[List[String], List[(String,Any)]] = {
  val n = name.trim  // remove whitespace
  if (n.length > 0 && n.matches("""^\p{Alpha}$""")) Success(List(key -> n))
  else Failure(List(s"Invalid $key <$n>"))
}

/** 验证字符串能否转换为大于0的整数。*/
def positive(key: String, n: String):
    Validation[List[String], List[(String,Any)]] = {
  try {
    val i = n.toInt
    if (i > 0) Success(List(key -> i))
    else Failure(List(s"Invalid $key $i"))
  } catch {
    case _: java.lang.NumberFormatException =>
      Failure(List(s"$n is not an integer"))
  }
}

def validateForm(firstName: String, lastName: String, age: String):
    Validation[List[String], List[(String,Any)]] = {
  validName("first-name", firstName) +++ validName("last-name", lastName) +++
    positive("age", age)
}

validateForm("Dean", "Wampler", "29")
// 返回值: Success(List((first-name,Dean), (last-name,Wampler), (age,29)))
validateForm("", "Wampler", "29")
// 返回值: Failure(List(Invalid first-name <>))
validateForm("D e a n", "Wampler", "29")
// 返回值: Failure(List(Invalid first-name <D e a n>))
validateForm("D1e2a3n_", "Wampler", "29")
// 返回值: Failure(List(Invalid first-name <D1e2a3n_>))
validateForm("Dean", "", "29")
// 返回值: Failure(List(Invalid last-name <>))
validateForm("Dean", "Wampler", "0")
// 返回值: Failure(List(Invalid age 0))
validateForm("Dean", "Wampler", "xx")
// 返回值: Failure(List(xx is not an integer))
validateForm("", "Wampler", "0")
// 返回值: Failure(List(Invalid first-name <>, Invalid age 0))
validateForm("Dean", "", "0")
//返回值: Failure(List(Invalid last-name <>, Invalid age 0))
validateForm("D e a n", "", "29")
// 返回值: Failure(List(Invalid first-name <D e a n>, Invalid last-name <>))
```

上面示例运用 scalaz.Validation 编写出了可用于对一组无关值进行验证的代码，这些代码美丽又简洁。如果这组值中包含了错误值，示例将返回所有被发现的错误；反之，则使用合适的数据结构将这些值集合在一起。

## 7.5 本章回顾与下一章提要

Either、Try 和 Validation 类型定义了程序的实际行为。Try 类型和 Validation 类型都期望能返回正确值。假如未返回正确值，这两种类型将会对你应该了解的错误信息进行封装。与之类似，Option 的类型签名很明确地表明了该类型对某一值能否出现的场景进行了封装。

使用这些类型能够减少对异常的使用[4]，同时我们还解决了一个重要的并发问题。由于我们无法保证异步执行的代码会运行在称为“调用者”（caller）的同一个线程内，因此调用者无法捕获其他代码所抛出的异常。不过，如果能够像返回正常值那样返回异常，调用者便能得到异常值。我们将在第 17 章深入讲述相关的细节。

你或许期望本章会对 Scala 语言花哨的 for 循环进行简单的讲解。其实不然，我们对 for 推导式进行了深入的研究，并学习了一组关于 for 推导式的强大工具。我们也学习了如何将 map、flatMap、foreach 以及 withFilter 这样的一组函数嵌入到 for 推导式中，并利用 for 推导式所提供的简洁、灵活并且强大的工具构建优秀的应用。

我们了解了如何使用 for 推导式对集合进行操作，而且我们还学会了如何将 for 推导式应用到其他的容器类型，尤其是 Option、util.Either、util.Try 和 scalaz.Validation 类型。

到此为止，我们已经完成了函数式编程的基础知识学习，并了解了 Scala 对函数式编程提供的支持。本书第 14 章和第 15 章将对类型系统进行讲解，我们也将在这两章中学到更多函数式编程的知识，而第 16 章则将深入讨论函数式编程的一些高级概念。

现在，我们将结束本章节的内容，转向 Scala 对面向对象编程的支持部分。其中许多相关的知识，我们已经在前面的章节中涉及过。

注 4：这里的“减少”一词意味着构造了一些具体的对象。我们用“减少”这个词来表达将某一概念封装到一个“正常”实例中的意思，因此我们可以像操作其他实例那样操作该对象。

第 8 章

# Scala面向对象编程

Scala 是一个函数式编程语言，也是一个面向对象的编程语言，与 Java、Python、Ruby、Smalltalk 等其他语言一样。直到现在才介绍 Scala 语言“面向对象的一面”有两个原因。

首先，我想强调的是，函数式编程已经成为解决现代编程问题的一项基本技能，这个技能对你而言可能是全新的。开始使用 Scala 时，人们很容易把它作为一个“更好的 Java”语言来使用，而忽略了它“函数式的一面”。

其次，Scala 在架构层面上提倡的方法是：小处用函数式编程，大处用面向对象编程。用函数式实现算法、操作数据，以及规范地管理状态，是减少 bug、压缩代码行数和降低项目延期风险的最好方法。另一方面，Scala 的 OO 模型提供很多工具，可用来设计可组合、可复用的模块。这对于较大的应用程序是必不可少的。因此，Scala 将两者完美地结合在了一起。

我假定你已经了解面向对象编程的基础知识，如 Java 的实现等。如果需要重新回顾，可以阅读 Robert C. Martin 的《敏捷软件开发》或者 Bertrand Meyer 的《面向对象软件构造》。如果你不熟悉设计模式，请参阅 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 共同编著的《设计模式》。他们四人被戏称为“四人组”（Gang of Four）。

在本章中，我们将快速回顾一下已经学习过的知识，并补充关于面向对象术语的其他细节，包括类声明和类继承的机制，价值类的概念，以及构造器在 Scala 中的工作原理。下一章我们将深入 trait，详细补充 Scala 对象模型的细节和标准库。

## 8.1 类与对象初步

类用关键字 class 声明，而单例对象用关键字 object 声明。出于这个原因，我使用“实例”一词指代一般对象。但实际上“实例”和“对象”在大多数 OO 语言中通常是同义的。

在类声明之前加上关键字 final，可以避免从一个类中派生出其他类。

abstract 关键字可以阻止类的实例化，例如：类中所包含或继承的成员声明（字段、方法或类型）没有提供具体实现的情况。即使类中并没有定义任何成员，我们仍然可以用 abstract 阻止类的实例化。

一个实例可以使用 this 关键字指代它本身。尽管在 Java 代码中经常看到 this 的这种用法，Scala 代码中却很少看到。原因之一是，Scala 中没有样板构造函数。考虑下面的 Java 代码：

```
// src/main/java/progscala2/basicoop/JPerson.java
package progscala2.basicoop;

public class JPerson {
  private String name;
  private int    age;

  public JPerson(String name, int age) {
    this.name = name;
    this.age  = age;
  }

  public void   setName(String name) { this.name = name; }
  public String getName()            { return this.name; }

  public void setAge(int age) { this.age = age;  }
  public int  getAge()        { return this.age; }
}
```

现在，将其与如下等价的 Scala 代码相比较。Scala 代码中没有任何样板代码。

```
class Person(var name: String, var age: Int)
```

在构造参数前加上 var，使得该参数成为类的一个可变字段，这在其他的 OO 语言中也称为实例变量或属性。在构造参数前加上 val，使得该参数成为类的一个不可变字段，用 case 关键字可以推断出 val，同时自动增加一些方法，如下所示：

```
case class ImmutablePerson(name: String, age: Int)
```

需要注意的是，实例的状态是实例所有字段内当前值的总和。

method（方法）指与实例绑定在一起的函数。换句话说，它的参数列表中有一个“隐含”的 this 参数。*方法*用关键字 def 定义。当其他函数或方法需要一个函数作为参数时，Scala 会自动将可用的方法“提升”为函数，作为前者的函数参数。

如同大多数静态类型语言一样，Scala 允许*方法重载*。只要各方法的完整签名是唯一的，

两个或更多方法就可以具有相同的名称。方法的签名包括返回类型，方法名称和参数类型的列表（参数的名称不重要）。因此，在 JVM 中只凭不同的返回类型不足以区分不同的方法。

然而也有例外。本书 5.2.5 节中，JVM 不允许某些方法完全不同。这是因为对于“高级类型”存在类型擦除机制，其中“高级类型”是包含类型参数的类型，如 List[A]。考虑下面的例子：

```
scala> object C {
     |   def m(seq: Seq[Int]): Unit = println(s"Seq[Int]: $seq")
     |   def m(seq: Seq[String]): Unit = println(s"Seq[String]: $seq")
     | }
<console>:9: error: double definition:
method m:(seq: Seq[String])Unit and
method m:(seq: Seq[Int])Unit at line 8
have same type after erasure: (seq: Seq)Unit
         def m(seq: Seq[String]): Unit = println(s"Seq[String]: $seq")
             ^
```

类型参数 Int 和 String 在二进制码中被擦除了。

不像 Java，Scala 可以用 type 关键字声明类型成员。正如我们在 2.13 节所见到的一样，这些类型成员是类型参数化的一种补充机制。它们经常被用来作为复杂类型的别名，以提供可读性。类型成员和参数化类型是不是两个重复的机制呢？不是，不过我们要到 14.5 节才来探究这个问题。

术语“成员”一词是类的字段、方法或类型的统称。与 Java 不同，如果方法有参数列表，该方法可以与类的字段同名：

```
scala> trait Foo {
     |    val x: Int
     |    def x: Int
     |  }
<console>:9: error: value x is defined twice
  conflicting symbols both originated in file '<console>'
         def x: Int
             ^
scala> trait Foo {
     |    val x: Int
     |    def x(i: Int): Int
     |  }
 defined trait Foo
```

类型名称必须唯一。

Scala 没有 Java 中的静态成员。但是 Scala 用 object 来保存多个实例共享的成员，如常量。

如果一个对象和一个类具有相同的名称，并在同一文件中定义，它们的关系就是伴随的。

回顾第 1 章可以知道，当一个对象和一个类具有相同的名称，且定义在同一文件时，它们相互伴随。对于 case 类，编译器自动生成一个伴随对象。

## 8.2 引用与值类型

Java 语法为 JVM 实现数据的方式提供了模型。首先，它提供了一组原生类型：`short`、`int`、`long`、`float`、`double`、`boolean`、`char`、`byte` 和关键字 `void`。它们被存储在堆栈中，或为了获得更好的性能，被存储于 CPU 寄存器。

其他的类型被称为引用类型，因为它们的所有实例都分配在堆中，引用这些实例的变量实际上指向了堆中的相应位置。不像 C 和 C++ 那样，栈上目前不存在任何“结构”类型的实例。Java 的未来版本正在考虑加入这种能力。所以，“引用类型”这个概念就是用来将这些实例同原生类型区分开的。引用类型的实例使用 `new` 关键字创建。

Scala 固然必须符合 JVM 的规则，但 Scala 做了改进，使得原生类型和引用类型的区别更明显。

所有引用类型都是 `AnyRef` 的子类型。`AnyRef` 是 `Any` 的子类型，而 `Any` 是 Scala 类型层次的根类型。所有值类型均为 `AnyVal` 的子类型，`AnyVal` 也是 `Any` 的子类型。`Any` 仅有这两个直接的子类型。需要注意，Java 的根类型 `Object`（http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html）实际上更接近 Scala 的 `AnyRef`，而不是 `Any`。

引用类型仍用 `new` 关键字来创建。类似其他不带参数的方法一样，如果构造器不带参数（在有的语言中称为“默认构造器”），我们在使用构造器时也可以去掉后面的括号。

Scala 沿用了 Java 中数字和字符串的字面量语法。例如：在 Scala 中，`val name = "Programming Scala"` 与 `val name = new String("Programming Scala")` 等价。不过，Scala 还为元组增加了字面量语法，`(1,2,3)` 就等价于 `new Tuple3(1,2,3)`。我们已经接触过 Scala 的一些语言特性，可以实现编译器原本不支持的字面量写法，如：用 `1 :: 2 :: 3 :: Nil` 表示 `Map("one" ->, "two" -> 2)`。

用带 `apply` 方法的对象创建引用类型的实例是很常见的做法，`apply` 方法起到工厂的作用（这种方法必须在内部调用 `new` 或对应的字面量语法）。由于 case 类会自动生成伴随对象及其 `apply` 方法，因此 case 类的实例通常就是用这种方法创建出来的。

`Short`、`Int`、`Long`、`Float`、`Double`、`Boolean`、`Char`、`Byte` 和 `Unit` 类型称为值类型，分别对应 JVM 的原型 `short`、`int`、`long`、`float`、`double`、`boolean`、`char`、`byte` 和 `void` 关键字。在 Scala 的对象模型中，所有的值类型均为 `AnyVal` 的子类型，`AnyVal` 是 `Any` 的两个子类型之一。

值类型的“实例”不是在堆上创建的。相反，Scala 用 JVM 原生类型来表示值类型，它们的值都存放在寄存器或栈上。值类型的“实例”总是用字面量来创建，如 `1`，`3.14`，`true`。`Unit` 对应的字面量是 `()`，不过我们很少显式地使用。

事实上，值类型没有构造器，所以像 `val I = new Int(1)` 这样的表达式将无法通过编译。

> **为什么 Unit 的字面量是 () 呢？**
>
> `Unit` 的行为与包含零个元素的元组很像，而元素个数为零的元组写为 `()`。Unit 这个名字来自数学里的乘法，任意值与单位值相乘，总是返回其原始值。这个单位值对于数字来说，就是 1。对于加法来说，0 为其单位值。我们在 16.1 节会再次涉及这个概念。

所以，Scala 最大限度地减少使用“包装过的”引用类型，为我们带来了两个体系各自的优点：原生类型的性能和源代码的对象语义。

语法的一致性允许我们声明值类型的参数化集合，如 `List[Int]`。与此相反，Java 则要求使用包装过的类型，如 `List<Integer>`。这样使得代码复杂化了。在 Java 的大数据库中，常常能见到一长串为原生类型定义的集合类型声明，如 `long` 和 `double`。你会见到一个用于表示 `long` 型向量的类，一个用于表示 `double` 型向量的类等。库的“当量”变得更大，其实现也不能做到代码重用。（目前，对集合和包装类型仍有讨论，可以参见 12.4 节。）

## 8.3 价值类

正如我们所看到的，Scala 中经常引入包装类型来实现新类型，这也被称为扩展方法（参见 5.4 节）。不幸的是，对值类型的包装，会将值类型变成引用类型，从而失去了原生类型的良好性能。

Scala 2.10 推出了一个解决方案，称为*价值类*（value class），和一个附带的特性，称为*通用特征*（universal trait）。这些类型限制了可声明的范围，但同样带来了好处：避免封装分配在堆上。

```
// src/main/scala/progscala2/basicoop/ValueClassDollar.sc

class Dollar(val value: Float) extends AnyVal {
  override def toString = "$%.2f".format(value)
}

val benjamin = new Dollar(100)
// 结果: benjamin: Dollar = $100.00
```

要成为一个有效的价值类，必须遵守以下的规则。

(1) 价值类有且只有一个公开的 `val` 参数（对于 Scala 2.11，该参数也可以是不公开的）。
(2) 参数的类型不能是价值类本身。
(3) 价值类被参数化时，不能使用 `@specialized`（http://www.scala-lang.org/api/current/scala/specialized.html）标记。
(4) 价值类没有定义其他构造器。
(5) 价值类只定义了一些方法，没有定义其他的 `val` 和 `var` 变量。
(6) 然而，价值类不能重载 `equals` 和 `hashCode` 方法。
(7) 价值类定义没有嵌套的特征、类或对象。
(8) 价值类不能被继承。

(9) 价值类只能继承自通用特征。
(10) 价值类必须是对象可引用的一个顶级类型或对象的一个成员。[1]

这是一个长长的清单，不过，当我们不遵守规则时，编译器会抛出错误消息。

本例中，Dollar 在编译时是一个外部类型。在运行时，该类型是被包装的类型，即 Float。

通常，被包装的类型是 AnyVal 的子类型之一，但并不是必须如此。如果换成引用类型，我们仍然可以受益于内存不在堆上分配的优势。例如，下例中，隐含了对电话号码字符串的包装：

```
// src/main/scala/progscala2/basicoop/ValueClassPhoneNumber.sc

class USPhoneNumber(val s: String) extends AnyVal {

  override def toString = {
    val digs = digits(s)
    val areaCode  = digs.substring(0,3)
    val exchange  = digs.substring(3,6)
    val subnumber = digs.substring(6,10)  //“客户编号”
    s"($areaCode) $exchange-$subnumber"
  }

  private def digits(str: String): String = str.replaceAll("""\D""", "")
}

val number = new USPhoneNumber("987-654-3210")
// 结果: number: USPhoneNumber = (987) 654-3210
```

价值类可以是一个 case 类，但是许多生成的额外方法和伴随对象不大可能被用到，导致产生的 class 文件就白白浪费了一些空间。

一个通用特征具有以下特性。

(1) 它可以从 Any 派生（而不能从其他通用特征派生）。
(2) 它只定义方法。
(3) 它没有对自身做初始化。

下面给出了一个改进版的 USPhoneNumber，这里混用了两个通用特征：

```
// src/main/scala/progscala2/basicoop/ValueClassUniversalTraits.sc

trait Digitizer extends Any {
  def digits(s: String): String = s.replaceAll("""\D""", "")         // ❶
}

trait Formatter extends Any {                                          // ❷
  def format(areaCode: String, exchange: String, subnumber: String): String =
    s"($areaCode) $exchange-$subnumber"
```

注 1：由于 Scala 丰富的类型系统，并非所有的类型都可以像在 Java 中那样，在正常的变量和方法声明中引用（不过，我们到目前为止看到的所有例子都可以正常进行引用）。在第 14 章中，我们将探索新的类型，并学习类型不能被引用的相关规则。

```
}

class USPhoneNumber(val s: String) extends AnyVal
    with Digitizer with Formatter {

  override def toString = {
    val digs = digits(s)
    val areaCode = digs.substring(0,3)
    val exchange = digs.substring(3,6)
    val subnumber  = digs.substring(6,10)
    format(areaCode, exchange, subnumber)                          // ❸
  }
}

val number = new USPhoneNumber("987-654-3210")
// 结果: number: USPhoneNumber = (987) 654-3210
```

❶ Digitizer 是一个通用特征，定义了我们之前的 digits 方法。

❷ Formatter 特征按我们想要的格式对电话号码进行格式化。

❸ 调用 Formatter.format。

Formatter 实际上解决了一个设计上的问题。我们可能要给 USPhoneNumber 指定另一个参数作为格式字符串，或需要一些机制去配置 toString 的格式，因为流行的格式可能有很多。但是，我们只允许传递一个参数给 USPhoneNumber。针对这种情况，我们可以在通用特征中去配置我们想要的格式！

然而，由于 JVM 的限制，通用特征有时会触发实例化（即实例的内存分配于堆中）。这里将需要实例化的情况总结如下。

(1) 当价值类的实例被传递给函数作参数，而该函数预期参数为通用特征且需要被实例实现。不过，如果函数的预期参数是价值类本身，则不需要实例化。
(2) 价值类的实例被赋值给数组。
(3) 价值类的类型被用作类型参数。

例如：当用 USPhoneNumber 调用以下方法时，我们会创建 USPhoneNumber 的一个实例：

```
def toDigits(d: Digitizer, str: String) = d.digits(str)
...
val digs = toDigits(new USPhoneNumber("987-654-3210"), "123-Hello!-456")
// 结果: digs: String = 123456
```

同样，当用 USPhoneNumber 调用以下参数化类型的方法时，也不得不产生 USPhoneNumber 的实例：

```
def print[T](t: T) = println(t.toString)
print(new USPhoneNumber("987-654-3210"))
// 结果: (987) 654-3210
```

总之，价值类提供了一个低开销的技术，用于定义扩展方法，并为类型定义有意义的领域名称（如 Dollar），这利用了被包装值的类型安全性。

“值类型”一词指代 Short、Int、Long、Float、Double、Boolean、Char、Byte 和 Unit 等 Scala 早就有的类型。而“价值类”一词指代继承 AnyVal 的自定义类。

有关价值类的实现细节的相关信息，请参阅 SIP-15：价值类（http://docs.scala-lang.org/sips/pending/value-classes.html）。SIP 表示 Scala 完善进行（scala improvement process），这是 Scala 社区提出的新的语言特性和库机制。

# 8.4 父类

子类是从父类或基类中派生的派生类，是大部分面向对象语言的核心特征。这种机制用来重用、封装和实现多态行为（具体行为取决于实例在类型层次结构中的实际类型）。

像 Java 一样，Scala 只支持单一继承，而不是多重继承。子类（或派生类）可以有且只有一个父类（即基类）。唯一的例外是，Scala 的类型结构中的根类 Any 没有父类。

我们已经见过父类及其子类的不少例子。下面我们从 2.13 节中抽取几个展示了类型成员用法的例子，并把其中的细节重点列出：

```
abstract class BulkReader {
  type In
  val source: In
  def read: String  // Read source and return a String
}

class StringBulkReader(val source: String) extends BulkReader {
  type In = String
  def read: String = source
}

class FileBulkReader(val source: java.io.File) extends BulkReader {
  type In = java.io.File
  def read: String = {...}
}
```

如在 Java 中一样，关键字 extend 表示后面是父类，因此本例中的父类为 BulkReader。在 Scala 中，当类继承 trait 时，也用 extend 表示（甚至当该类用 with 关键字混入了其他 trait 时也是如此）。此外，当 trait 是其他 trait 或类的子 trait 时，也用 extend。是的，trait 可以继承类。

如果我们不指定父类，默认父类为 AnyRef。

# 8.5 Scala的构造器

Scala 将主构造器与零个或多个辅助构造器区分开，辅助构造器也被称为次级构造器。在 Scala 中，主构造器是整个类体。构造器所需的所有参数都被罗列在类名称后面。StringBulkReader 和 FileBulkReader 就是两个例子。

让我们重温几个在第 5 章碰到的简单 case 类，Address 和 Person。考虑使用次级构造器进行改进：

```
// src/main/scala/progscala2/basicoop/PersonAuxConstructors.scala
package progscala2.basicoop

case class Address(street: String, city: String, state: String, zip: String) {

  def this(zip: String) =                                        // ❶
    this("[unknown]", Address.zipToCity(zip), Address.zipToState(zip), zip)
}

object Address {

  def zipToCity(zip: String)  = "Anytown"                        // ❷
  def zipToState(zip: String) = "CA"
}

case class Person(
  name: String, age: Option[Int], address: Option[Address]) {    // ❸

  def this(name: String) = this(name, None, None)                // ❹

  def this(name: String, age: Int) = this(name, Some(age), None)

  def this(name: String, age: Int, address: Address) =
    this(name, Some(age), Some(address))

  def this(name: String, address: Address) = this(name, None, Some(address))
}
```

❶ 次级构造器只带一个参数，即邮政编码。内部调用了其他辅助函数，通过邮政编码得到城市名和州名，但无法得到街道名。

❷ 通过邮政编码查找城市和州的辅助函数（至少假设该辅助函数能做到这一点）。

❸ 使得年龄和地址成为可选参数。

❹ 提供次级构造器的便利接口，让用户指定部分或全部参数值。

需要注意的是，辅助构造被命名为 this，它的第一个表达式必须调用主构造器或其他辅助构造器。编译器还要求被调用的构造器在代码中先于当前构造器出现。所以，我们在代码中必须小心地排列构造器的顺序。

通过强制让所有构造器最终都调用主构造器，可以将代码冗余最小化，并确保新实例的初始化逻辑的一致性。

Address 的次级构造器包含了一些具体的有用逻辑，这不同于 Person 的次级构造器，Person 的初次构造器只是多提供了几个可调用的便利函数。

上述文件是用 sbt 编译的，所以我们可以在下面的脚本中使用其中的类型：

```
// src/main/scala/progscala2/basicoop/PersonAuxConstructors.sc
import progscala2.basicoop.{Address, Person}
```

```
val a1 = new Address("1 Scala Lane", "Anytown", "CA", "98765")
// 结果: Address(1 Scala Lane,Anytown,CA,98765)

val a2 = new Address("98765")
// 结果: Address([unknown],Anytown,CA,98765)

new Person("Buck Trends1")
// 结果: Person(Buck Trends1,None,None)

new Person("Buck Trends2", Some(20), Some(a1))
// 结果: Person(Buck Trends2,Some(20),
//          Some(Address(1 Scala Lane,Anytown,CA,98765)))

new Person("Buck Trends3", 20, a2)
// 结果: Person(Buck Trends3,Some(20),
//          Some(Address([unknown],Anytown,CA,98765)))

new Person("Buck Trends4", 20)
// 结果: Person(Buck Trends4,Some(20),None)
```

虽然此代码运行得很好，但实际上还是存在一些问题。首先，Person 有很多参数类似的次级构造器。事实上我们可以对使用默认值的方法参数进行定义，而且用户也可以在调用方法时命名参数。

重新考虑 Person 这个类型。首先，我们来为 age 和 address 添加默认值，并且假设用户对于需要使用 Some(…) 作为参数的情况并不觉得繁复累赘：

```
// src/main/scala/progscala2/basicoop/PersonAuxConstructors2.sc
import progscala2.basicoop.Address

val a1 = new Address("1 Scala Lane", "Anytown", "CA", "98765")
val a2 = new Address("98765")

case class Person2(
  name: String,
  age: Option[Int] = None,
  address: Option[Address] = None)

new Person2("Buck Trends1")
// 结果: Person2 = Person2(Buck Trends1,None,None)

new Person2("Buck Trends2", Some(20), Some(a1))
// 结果: Person2(Buck Trends2,Some(20),
//       Some(Address(1 Scala Lane,Anytown,CA,98765)))

new Person2("Buck Trends3", Some(20))
// 结果: Person2(Buck Trends3,Some(20),None)

new Person2("Buck Trends4", address = Some(a2))
// 结果: Person2(Buck Trends4,None,
//        Some(Address([unknown],Anytown,CA,98765)))
```

虽然 Person 的调用者多写了一点点代码，但是对于库的开发者而言维护负担的显著下降是件好事。这也算是一种平衡。

我们决定使用减小用户负担的那种选择。第二个问题是，在我们的实现中，用户必须使用 new 来创建实例。也许你已经注意到了，实例中就是用 new 来构造实例的。

如果尝试把 new 关键字去掉，会发生什么呢？除非调用的是主构造器，不然你会得到一个编译错误。



编译器不会自动为 case 类的次级构造器创建 apply 方法。

但是，如果我们在伴随对象中重载 Person.apply，就可以得到便利的“构造器”，避免使用 new。以下是 Person 的最终实现方式，命名为 Person3：

```
// src/main/scala/progscala2/basicoop/PersonAuxConstructors3.scala
package progscala2.basicoop3
import progscala2.basicoop.Address

case class Person3(
  name: String,
  age: Option[Int] = None,
  address: Option[Address] = None)

object Person3 {

  // 由于我们重载的是普通方法,而不是构造器,
  // 所以必须明确地指定返回类型,在这里返回类型是Person3。
  def apply(name: String): Person3 = new Person3(name)

  def apply(name: String, age: Int): Person3 = new Person3(name, Some(age))

  def apply(name: String, age: Int, address: Address): Person3 =
    new Person3(name, Some(age), Some(address))

  def apply(name: String, address: Address): Person3 =
    new Person3(name, address = Some(address))
}
```

注意，与构造器不同，apply 这样的重载方法必须有明显的返回类型注释。

最后，给出一个使用最终版本 Person 的脚本：

```
// src/main/scala/progscala2/basicoop/PersonAuxConstructors3.sc
import progscala2.basicoop.Address
import progscala2.basicoop3.Person3

val a1 = new Address("1 Scala Lane", "Anytown", "CA", "98765")
val a2 = new Address("98765")

Person3("Buck Trends1")                              // 主
// 结果: Person3(Buck Trends1,None,None)

Person3("Buck Trends2", Some(20), Some(a1))          // 主
// 结果: Person3(Buck Trends2,Some(20),
```

```
//           Some(Address(1 Scala Lane,Anytown,CA,98765)))

Person3("Buck Trends3", 20, a1)
// 结果: Person3(Buck Trends3,Some(20),
//           Some(Address(1 Scala Lane,Anytown,CA,98765)))

Person3("Buck Trends4", Some(20))                          // 主
// 结果: Person3(Buck Trends4,Some(20),None)

Person3("Buck Trends5", 20)
// 结果: Person3(Buck Trends5,Some(20),None)

Person3("Buck Trends6", address = Some(a2))                // 主
// 结果: Person3(Buck Trends6,None,
//           Some(Address([unknown],Anytown,CA,98765)))

Person3("Buck Trends7", address = a2)
// 结果: Person3(Buck Trends7,None,
//           Some(Address([unknown],Anytown,CA,98765)))
```

所有注释为“主”的调用均调用了 `Person3` 这个 `case` 类自动生成的主 `apply` 方法。其他没有该注释的地方，调用的是其他 `apply` 方法。

事实上，在 Scala 代码中定义次级构造器并不是很常见。因为还有其他替代的技术，它们通常为用户提供同样灵活的构造选项，同时减少样板代码。注意要合理使用 Scala 中的命名参数和可选参数，并科学地使用对象中重载的 `apply`“工厂”方法。

# 8.6 类的字段

在本章的开头我们曾提醒大家，如果在主构造函数参数的前面加上 `val` 或 `var` 关键字，该参数就成为实例的一个字段。对于 case 类，`val` 是默认的。这样可以大大减少冗余的代码，但是它是如何转化成字节码的呢？

其实，不过是 Scala 偷偷地干了 Java 代码中明显做的事情。类会创建一个私有的字段，并生产对应的 getter 和 setter 访问方法。考虑下面这个简单的 Scala 类：

```
class Name (var value: String)
```

从概念上讲，上述代码和下面的代码是等价的：

```
class Name(s: String) {
  private var _value: String = s                          // ❶

  def value: String = _value                              // ❷

  def value_=(newValue: String): Unit = _value = newValue  // ❸
}
```

❶ 不可见的字段，在本例中声明为可变变量。

❷ getter，即读方法。

❸ setter，即写方法。

注意 value_= 这个方法名的一般规范。当编译器看到这样的一个方法名时，它会允许客户端代码去掉下划线 _，转而使用中缀表达式，这就好像我们是在设置对象的一个裸字段一样：

```
scala> val name = new Name("Buck")
name: Name = Name@2aed6fc8

scala> name.value
res0: String = Buck

scala> name.value_=("Bubba")
name.value: String = Bubba

scala> name.value
res1: String = Bubba

scala> name.value = "Hank"
name.value: String = Hank

scala> name.value
res2: String = Hank
```

如果我们用 val 关键字声明一个不可变字段，那就不会自动生成写方法，只会生成读方法。

如果你想在读方法或写方法内实现自定义逻辑，可以按以下规范来实现。

对于非 case 类，如果我们向构造器传递参数时不希望参数成为类的字段，可以在构造器中省略 val 或 var。例如，我们虽然需要向构造器传递一个参数，但却希望构造器退出后丢弃它。

注意，该值仍然在类体的作用范围内。正如前文有关隐式转换的例子中我们看到的那样，它们仍然指向构造实例时所用的参数，但这些参数大多没有被声明为类的字段。回顾一下 5.2.7 节的 Pipeline 示例：

```
object Pipeline {
  implicit class toPiped[V](value:V) {
    def |>[R] (f : V => R) = f(value)
  }
}
```

尽管在 |> 方法中，toPiped 引用了 value 变量，但 value 并不是类 toPiped 的一个字段。无论构造器参数有没有用 val 或 var 声明，该参数在整个类体中都是可见的。所以，该参数可以被类的成员使用，如类的方法成员。对比 Java 或其他 OO 语言中定义的构造器，我们得知：由于这些构造器是一个方法，所以传递给构造器的参数在方法外是不可见的。由此可见，无论是显式地还是隐式地，构造器的参数都必须像字段一样被“保存”起来。

为什么不总是将构造器的参数变成字段呢？因为字段对类的客户端是可见的（也就是说，除非被声明为私有的或保护的，客户端都可以看到字段。我们将在第 13 章讨论这一点）。而这些构造器的参数只有在确实需要暴露给用户状态的情况下，才会变成字段；否则它们不应该成为字段，而是属于类体私有的。

### 8.6.1 统一访问原则

你可能会对 Scala 没有遵循 JavaBeans 的约定规范，也就是把 `value` 字段的读方法和写方法分别命名为 `getValue` 和 `setValue`，感到疑惑不解。事实上，Scala 遵循统一访问的原则。

正如我们在 `Name` 这个例子中看到的，客户端似乎可以不经过方法就对“裸”字段值进行读和写的操作，但实际上它们调用了方法。另一方面，我们可以用默认的公开可见性声明一个字段，然后像裸字段一样访问该字段：

```
class Name2(s: String) {
  var value: String = s
}
```

现在，`value` 是一个公开字段，没有访问方法。

我们来试试看：

```
scala> val name2 = new Name2("Buck")
name2: Name2 = Name2@303becf6

scala> name2.value
res0: String = Buck

scala> name2.value_=("Bubba")
name2.value: String = Bubba

scala> name2.value
res1: String = Bubba
```

请注意，用户的“体验”是一致的。用户代码不了解实现，这使我们可以在需要的时候，自由地将直接操作裸字段改为通过访问方法来操作。例如：我们要在写操作中添加某些验证工作，或者为了提高效率，在读取某个结果时采用惰性求值。这些情况下，我们可以通过访问方法来操作裸字段。相反地，我们也可以用公开可见性的字段代替访问方法，以消除该方法调用的开销（尽管 JVM 可能会消除这种开销）。

因此，统一访问原则的重要益处在于，它能最大程度地减少客户端代码对类的内部实现所必须了解的知识。尽管重新编译仍然是必需的，我们可以改变具体实现，而不必强制客户端代码跟着做出改变。

Scala 实现统一访问原则的同时，没有牺牲访问控制功能，并且满足了在简单的读写基础上增加其他逻辑的需求。

> Scala 没有采用 Java 风格的 getter 和 setter 方法，而是支持统一访问原则。在统一访问原则中，读写“裸”字段的语法和通过间接调用方法实现读写的语法是一样的。

然而有时为了与 Java 库交互，也会需要 JavaBeans 风格的访问方法。这时可以用 `scala.reflect.BeanProperty`（http://www.scala-lang.org/api/current/scala/beans/BeanProperty.html）或 `BooleanBeanProperty`（http://www.scala-lang.org/api/current/scala/beans/BooleanBean

Property.html）给类做标记。更多细节，见 22.3 节。

### 8.6.2　一元方法

我们已经看到，编译器允许我们为字段 foo 定义赋值方法 foo_=，然后使用简便的 myinstance.foo = value。另一种操作符——一元操作符，我们还不知道如何实现。

取反就是一个例子。如果我们要实现一个复数类，该如何支持实例的相反数（如 c 的相反数 -c）呢？请看下面的代码：

```
// src/main/scala/progscala2/basicoop/Complex.sc

case class Complex(real: Double, imag: Double) {
  def unary_- : Complex = Complex(-real, imag)                       // ❶
  def -(other: Complex) = Complex(real - other.real, imag - other.imag)
}

val c1 = Complex(1.1, 2.2)
val c2 = -c1                              // Complex(-1.1, 2.2)
val c3 = c1.unary_-                       // Complex(-1.1, 2.2)
val c4 = c1 - Complex(0.5, 1.0)           // Complex(0.6, 1.2)
```

❶ 方法名为 unary_X，这里 X 就是我们想要使用的操作符。在本例中，X 就是 -。注意 - 和 : 之间的空格，空格在这里是必须的，它可以告诉编译器方法名以 - 而不是 : 结尾！为了比较，我们也实现了常见的减号操作符。

我们一旦定义了一元操作符，就可以将操作符放在实例之前，就像我们在定义 c2 时所做的那样。也可以像定义 c3 时那样，将一元操作符当做其他方法一般进行调用。

## 8.7　验证输入

如果我们想验证输入的参数，以确保产生的实例处于有效状态，该怎么做呢？Predef（http://www.scala-lang.org/api/current/index.html#scala.Predef$）定义了一系列名为 require 的重载方法，可以实现这一目的。考虑以下这个封装了美国邮政编码的类。邮政编码允许使用两种形式，5 位数字或者“邮政编码 +4 位数字”的形式。后者较前者多了结尾的 4 位数字，常被写为类似“12345-6789”的形式。另外，不是所有的数字都对应有真实的邮编：

```
// src/main/scala/progscala2/basicoop/Zipcode.scala
package progscala2.basicoop

case class ZipCode(zip: Int, extension: Option[Int] = None) {
  require(valid(zip, extension),                                     // ❶
    s"Invalid Zip+4 specified: $toString")

  protected def valid(z: Int, e: Option[Int]): Boolean = {
    if (0 < z && z <= 99999) e match {
      case None    => validUSPS(z, 0)
      case Some(e) => 0 < e && e <= 9999 && validUSPS(z, e)
    }
    else false
```

```
  }

  /**这是个有效的美国邮政编码吗？ */
  protected def validUSPS(i: Int, e: Int): Boolean = true          // ❷

override def toString =                                            // ❸
   if (extension != None) s"$zip-${extension.get}" else zip.toString
}

object ZipCode {
  def apply (zip: Int, extension: Int): ZipCode =
    new ZipCode(zip, Some(extension))
}
```

❶ 使用 `require` 方法验证输入。

❷ 真正的方法实现应该查询 USPS 认可的数据库来验证邮政编码是否确实存在。

❸ 覆写 `toString` 方法，返回符合人们预期的邮政编码格式，对结尾可能的四位数字进行恰当的处理。

以下是调用它的脚本：

```
// src/main/scala/progscala2/basicoop/Zipcode.sc
import progscala2.basicoop.ZipCode

ZipCode(12345)
// 结果: ZipCode = 12345

ZipCode(12345, Some(6789))
// 结果: ZipCode = 12345-6789

ZipCode(12345, 6789)
// 结果: ZipCode = 12345-6789

try {
  ZipCode(0, 6789)  // Invalid Zip+4 specified: 0-6789
} catch {
  case e: java.lang.IllegalArgumentException => e
}

try {
  ZipCode(12345, 0)  // Invalid Zip+4 specified: 12345-0
} catch {
  case e: java.lang.IllegalArgumentException => e
}
```

定义 `ZipCode` 这种领域专用的类的充分理由是：这种类可以在构造时对值的有效性做一次检验，然后类 `ZipCode` 的使用者就不再需要再次检验了。

`Predef` 中的 `ensuring` 和 `assume` 等方法也用于实现类似的目的。我们将在 23.5 节探索 `require` 和这两个断言方法的更多用途。

虽然我们在构造器的背景下讨论输入的验证，但实际上我们也可以在任何方法中调用这些断言方法。然而，价值类的类体是一个例外，它不能使用断言验证，否则就需要调用分配

堆。不过，由于 ZipCode 带有两个构造器参数，它无论如何不会是价值类。

## 8.8 调用父类构造器（与良好的面向对象设计）

派生类的主构造器必须调用父类的构造器，可以是父类的主构造器或次级构造器。在以下示例中，Employee 是 Person 的子类：

```
// src/main/scala/progscala2/basicoop/EmployeeSubclass.sc
import progscala2.basicoop.Address

case class Person(    // This was Person2 previously, now renamed.
  name: String,
  age: Option[Int] = None,
  address: Option[Address] = None)

class Employee(                                                      // ❶
  name: String,
  age: Option[Int] = None,
  address: Option[Address] = None,
  val title: String = "[unknown]",                                   // ❷
  val manager: Option[Employee] = None) extends Person(name, age, address) {

  override def toString =                                            // ❸
    s"Employee($name, $age, $address, $title, $manager)"
}

val a1 = new Address("1 Scala Lane", "Anytown", "CA", "98765")
val a2 = new Address("98765")

val ceo = new Employee("Joe CEO", title = "CEO")
// 结果: Employee(Joe CEO, None, None, CEO, None)

new Employee("Buck Trends1")
// 结果: Employee(Buck Trends1, None, None, [unknown], None)

new Employee("Buck Trends2", Some(20), Some(a1))
// 结果:  Employee(Buck Trends2, Some(20),
//            Some(Address(1 Scala Lane,Anytown,CA,98765)), [unknown], None)

new Employee("Buck Trends3", Some(20), Some(a1), "Zombie Dev")
// 结果:  Employee(Buck Trends3, Some(20),
//            Some(Address(1 Scala Lane,Anytown,CA,98765)), Zombie Dev, None)

new Employee("Buck Trends4", Some(20), Some(a1), "Zombie Dev", Some(ceo))
// 结果:  Employee(Buck Trends4, Some(20),
//            Some(Address(1 Scala Lane,Anytown,CA,98765)), Zombie Dev,
//            Some(Employee(Joe CEO, None, None, CEO, None)))
```

❶ Employee 被声明为一个普通的类，而不是 case 类。在下文中我会解释这么做的原因。

❷ 新的字段 title 和 manager 需要用 val 关键字声明，因为 Employee 不是 case 类。其他参数是类从 Person 继承的字段。注意，我们也调用了 Person 的主构造器。

❸ 覆写 toString 方法。如果不覆写，将会调用 Person.toString。

在 Java 中，我们会定义构造方法，并用 `super` 调用父类的初始化逻辑。而 Scala 中，我们用 `ChildClass(…) extends ParentClass(…)` 的语法隐式地调用父类的构造器。

> 尽管像在 Java 中一样，`super` 可以用来调用被覆写的父类方法，但它不能用来调用父类的构造器。

## 良好的面向对象设计：题外话

这段代码有“异味”。`Employee` 的声明把带 `val` 关键字的参数和不带关键字的参数混杂在参数列表中。但更深层次的问题潜伏在这段代码背后。

我们可以从一个 case 类派生出一个非 case 类，或者反过来。但我们无法从一个 case 类派生出另一个 case 类。这是因为，自动生成的 `toString`、`equals` 和 `hashCode` 方法在子类中无法正常工作。这意味着它们忽视了这种可能性，即实例实际上可以是 case 类的子类。

这种问题实际上是设计上的问题。它反映了子类继承的问题。例如：具有相同姓名、年龄、地址的 `Employee` 类的实例和 `Person` 类的实例是否应该被视为等价的呢？如果对对象平等做宽松地解释，我们认为它们是相等价的；如果更严格地解释，我们则会说它们不相等。事实上，相等的数学定义需要满足交换律：`somePerson == someEmployee` 与 `someEmployee == somePerson` 应该返回相同的结果。灵活的等价解释将打破这种交换律，因为你绝不会想到一个 `Employee` 实例会与一个不是 `Employee` 的 `Person` 实例相等。

实际上，在这里，`equals` 的问题要更糟糕，因为 `Employee` 没有覆写 `equals` 和 `hashCode` 方法。我们实际上是将所有 `Employee` 的实例当作 `Person` 的实例来对待的。

这对于小的类型而言是非常危险的。不可避免地会有人创建 `Employee` 集合，在集合中对元素进行排序，或将元素作为散列映射的键。由于这会分别调用 `Person.equals` 和 `Person.hashCode`，所以当出现两个人同名为 John Smith，一人是 CEO，另一人在收发室工作时，就会出现异常行为。混淆两个人的场景经常发生，但还没有经常到能够很容易地复现以帮助修复这个 bug 的程度！

真正的问题是，我们继承了状态属性。也就是说，我们在本例中使用继承来添加状态属性 `title` 和 `manager`。相反，继承相同状态属性的行为能够更容易地实现。这避免了刚刚描述的 `equals` 和 `hashCode` 的问题。

当然，继承机制的问题存在已久。如今，好的面向对象设计更青睐于组合，而不是继承。因此，我们选择将功能单元组合起来，而不是构建一个类型继承结构。

我们在下一章将看到：trait 使得 Scala 的组合比 Java 的接口更容易使用，至少在 Java 8 之前是这样。在这本书中，不属于“玩具”性质的实例将不会使用继承来增加状态。幸运的是，继承在 Scala 库中也很罕见。

因此，Scala 的团队可以选择实现对子类友好的 `equals`、`hashCode` 和 `toString`，但这在支持糟糕设计时会增加额外的复杂度。case 类为方便且简单的域类型提供了模式匹配和实例

分解，但支持继承结构并不是它们的目的。

当使用继承时，建议遵循以下规则。

(1) 一个抽象的基类或 trait，只被下一层的具体的类继承，包括 case 类。
(2) 具体类的子类永远不会再次被继承，除了两种情况：a. 类中混入了定义于 trait（见第 9 章）中的其他行为。理想情况下，这些行为应该是正交的，即不重叠的。b. 只用于支持自动化单元测试的类。
(3) 当使用子类继承似乎是正确的做法时，考虑将行为分离到 trait 中，然后在类里混入这些 trait。
(4) 切勿将逻辑状态跨越父类和子类。

对最后一条中的“逻辑”，我指的是，我们可能有一些私有的，专门实现的状态。这种状态不影响外部可见性、相等、散列等逻辑。例如：我们的库中可能有某些集合类型的子类，增加了私有的字段，用于实现缓存或日志功能（此时通过混入 trait 实现该特性并不是一个好的选择）。

那么，如何实现 `Employee` 呢？如果通过集成 `Person` 创建 `Employee` 不太好，那我们又该怎么做呢？答案取决于使用的场景。如果我们正在实现一个人力资源的应用程序，我们是否还需要一个单独的 `Person` 概念？还是直接将 `Employee` 声明为 `case` 类作为基类使用呢？进一步讲，我们是否需要为此提供任何类型？如果我们正在处理从数据库查询得到的结果，只用元组或其他容器来保存从查询返回的字段是否就足够了？我们是否可以省掉必须声明一个类型的“仪式”？

假设我们的确需要区分 `Person` 和 `Employee` 的概念。以下是我的一种实现方式：

```
// src/main/scala/progscala2/basicoop/PersonEmployeeTraits.scala
package progscala2.basicoop2                                           // ❶

case class Address(street: String, city: String, state: String, zip: String)

object Address {
  def apply(zip: String) =                                             // ❷
    new Address(
      "[unknown]", Address.zipToCity(zip), Address.zipToState(zip), zip)

  def zipToCity(zip: String)  = "Anytown"
  def zipToState(zip: String) = "CA"
}

trait PersonState {                                                    // ❸
  val name: String
  val age: Option[Int]
  val address: Option[Address]

  // 在这定义一些公共的方法?
}

case class Person(                                                     // ❹
  name: String,
  age: Option[Int] = None,
```

```
  address: Option[Address] = None) extends PersonState

trait EmployeeState {                                    // ❺
  val title: String
  val manager: Option[Employee]
}

case class Employee(                                     // ❻
  name: String,
  age: Option[Int] = None,                               // ❼
  address: Option[Address] = None,
  title: String = "[unknown]",
  manager: Option[Employee] = None)
extends PersonState with EmployeeState
```

❶ 因为这些类型的早期版本已经定义在包 `oop` 中了，所以这里用了其他包名。

❷ 此前，`Address` 有一个次级构造器。现在我们用一个次级工厂方法。

❸ 为 `Person` 拥有的状态（我们希望的状态）定义一个 trait。你可以挑一个比 `PersonState` 更好的命名。

❹ 当只有 `Person` 实例时，使用这个 case 类来实现 `PersonState`。

❺ 对 `Employee` 采用相同的做法，尽管在这里声明单独的 trait 和 case 类没有那么大用处。虽然一致性增加了引入单独的 trait 和 case 类的“仪式”，但保持一致性有其可取之处。

❻ `Employee` case 类。

❼ 请注意，我们必须为 `Person` 和 `Employee` 共享的字段定义两次默认值。这是一个小缺点（除非我们真的需要使用不同的默认值）。

需要注意，`Employee` 不再是 `Person` 的一个子类，但它是 `PersonState` 的一个子类，因为它混入了该 trait。另外，`EmployeeState` 也不再是 `PersonState` 的子类。图 8-1 中的类图说明了类间的关系。

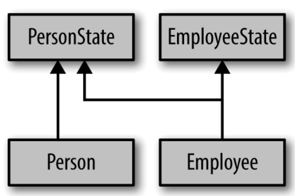



图 8-1：`PersonState`、`Person`、`EmployeeState` 和 `Employee` 的类图

注意到，`Person` 和 `Employee` 都混入了 trait，但 `Employee` 并非从其他具体类中派生。

我们来试着创建几个对象：

```
// src/main/scala/progscala2/basicoop/PersonEmployeeTraits.sc
import progscala2.basicoop.{ Address, Person, Employee }

val ceoAddress  = Address("1 Scala Lane", "Anytown", "CA", "98765")
```

```
// 结果: ceoAddress: oop2.Address = Address(1 Scala Lane,Anytown,CA,98765)

val buckAddress = Address("98765")
// 结果: buckAddress: oop2.Address = Address([unknown],Anytown,CA,98765)

val ceo = Employee(
  name = "Joe CEO", title = "CEO", age = Some(50),
  address = Some(ceoAddress), manager = None)
// 结果: ceo: oop2.Employee = Employee(Joe CEO,Some(50),
//            Some(Address(1 Scala Lane,Anytown,CA,98765)),CEO,None)

val ceoSpouse = Person("Jane Smith", address = Some(ceoAddress))
// 结果: ceoSpouse: oop2.Person = Person(Jane Smith,None,
//            Some(Address(1 Scala Lane,Anytown,CA,98765)))

val buck = Employee(
  name = "Buck Trends", title = "Zombie Dev", age = Some(20),
  address = Some(buckAddress), manager = Some(ceo))
// 结果: buck: oop2.Employee = Employee(Buck Trends,Some(20),
//             Some(Address([unknown],Anytown,CA,98765)),Zombie Dev,
//             Some(Employee(Joe CEO,Some(50),
//               Some(Address(1 Scala Lane,Anytown,CA,98765)),CEO,None)))

val buckSpouse = Person("Ann Collins", address = Some(buckAddress))
// 结果: buckSpouse: oop2.Person = Person(Ann Collins,None,
//             Some(Address([unknown],Anytown,CA,98765)))
```

你会发现，我在好几个声明中都用了命名参数。当一个构造函数器或其他方法有很多参数时，我喜欢使用命名参数，以清楚地表明每个参数的意思。当好几个参数为同一类型时，bug 可以很容易地被避免，同时值之间也方便切换。当然，你应该通过减少参数个数，确保参数的类型唯一，来避免这些风险。

现在，你对 trait 的兴趣已经被激起。我们会在后面的章中深度地讨论它。现在，我们需要讨论面向对象的最后一个话题。

## 8.9 嵌套类型

Scala 允许我们嵌套类型的成名和定义。例如：在对象中定义类型转义的异常和其他有用的类型，就是很常见的做法。以下是一个数据库层可能的骨架结构：

```
// src/main/scala/progscala2/basicoop/NestedTypes.scala

object Database {                                                // ❶
  case class ResultSet(/*...*/)                                  // ❷
  case class Connection(/*...*/)                                 // ❸

  case class DatabaseException(message: String, cause: Throwable) extends
    RuntimeException(message, cause)

  sealed trait Status                                            // ❹
  case object Disconnected extends Status
  case class  Connected(connection: Connection)  extends Status
```

```
  case class  QuerySucceeded(results: ResultSet) extends Status
  case class  QueryFailed(e: DatabaseException)  extends Status
}

class Database {
  import Database._

  def connect(server: String): Status = ???                        // ❺
  def disconnect(): Status = ???

  def query(/*...*/): Status = ???
}
```

❶ 数据库的一个简单接口。

❷ 封装了查询结果的集合。我们把不关心的细节省略了。

❸ 封装了连接池和其他信息。

❹ 使用 sealed 的继承结构表示状态；所有允许的值都在这里定义。当实例实际上不携带状态信息时，使用 case 对象。这些对象表现得像“标志位”，用来表示状态。

❺ `???` 是定义在 `Predef` 中的真实方法。它会抛出一个异常，用来表示某一方法尚未实现的情况。该方法是最近才引入 Scala 库的。

> 当 case 类没有用任何字段表示状态信息时，考虑使用 case 对象。
>
> 当方法还正处在开发阶段时，`???` 方法作为占位符十分有用。因为这样可以使代码通过编译。问题是你没法调用那个方法！

关于 case 对象，我发现了一个“漏洞”：

```
scala> case object Foo
defined object Foo

scala> Foo.hashCode
res0: Int = 70822

scala> "Foo".hashCode
res1: Int = 70822
```

显然，为 case 对象生成的 `hashCode` 方法仅仅将对象的名称做了散列。而对象的包像对象的字段一样被忽略了。这意味着，当需要更强的 `hashCode` 方法时，case 对象是存在风险的。

> 当需要更强的 `hashCode` 方法时，避免使用 case 对象，例如：对象是基于散列运算的映射或集合的键。

## 8.10 本章回顾与下一章提要

我们补充上了关于 Scala 的对象模型的基础知识，包括构造、继承和类型的嵌套。有时我们甚至把话题扯远，讨论到在 Scala 或任何其他语言中，什么是良好的面向对象的设计问题。

我们还讨论了 trait，这是 Scala 对 Java 接口的增强版。trait 是组合各部分行为的有力工具，它避免了使用继承及继承带来的缺点。在下一章中，我们将完成对 trait 的理解，并掌握如何使用 trait 来解决各种设计问题。

第 9 章

# 特征

在 Java 中，类可以实现任意数量的接口。这种模型非常适用于声明实现了多个抽象的类。不过，这类模型也存在一个明显的缺点。对于一些接口而言，使用该接口的所有类使用了样板代码实现接口的大量功能。

通常，接口中定义了一些与实现类中的其他成员无关联的成员（这些成员具有“正交性”）。而混入（mixin）一词便适用于这类聚焦的、可重用的状态和行为。理想情况下，我们应单独维护这些公用的行为，而不应该依赖于任何使用这些行为的具体类型。

在 Java 8 诞生之前，Java 未提供用于定义和使用这类可重用代码的内置机制。为此，Java 程序员必须使用特定的方法进行复用某一接口的实现代码。最坏的情况下，开发人员必须将相同的代码复制粘贴到所有需要这一功能的类中。这里存在一个略好一些但谈不上完美的解决方案：单独编写一个实现了该行为的类，原始类中会保存一份该类的示例并负责为支持类提供代理方法调用。这一方案能够解决代码复用的问题，但也添加了一些没有必要的额外开销，同时容易导致错误的样板代码。

## 9.1 Java 8中的接口

Java 8 做出了改变。现在我们可以在接口中定义方法，这些方法被称为 defender 方法或默认方法。实现类仍可以提供自己的实现。如果实现类未提供自己的实现的话，defender 方法会被调用。因此，Java 8 中的接口行为更接近于 Scala 中的 trait。

但是，Java 8 中的接口与 Scala 中的 trait 仍有不同之处。Java 8 中的接口只能定义静态字段，而 Scala 中的 trait 则可以定义实例级字段。这意味着 Java 8 中的接口无法管理实例状态。接口实现类必须提供字段以记录状态。这也意味着 defender 方法无法访问接口实现体的状态信息，从而限制了 defender 方法的用途。

在后面的学习中，请思考如何使用 Java 8 中的接口和类来实现 trait 和类。什么样的代码能够简单地迁移到 Java 8，而什么样的代码又不能呢？

## 9.2 混入trait

为了能够理解 trait 在模块化 Scala 代码的作用，我们首先学习一下下面的这段代码。图形用户界面（GUI）中的按钮会使用这段代码，当点击事件发生时，这段代码会通过回调机制通知客户：

```
// src/main/scala/progscala2/traits/ui/ButtonCallbacks.scala
package progscala2.traits.ui

class ButtonWithCallbacks(val label: String,
    val callbacks: List[() => Unit] = Nil) extends Widget {

  def click(): Unit = {
    updateUI()
    callbacks.foreach(f => f())
  }

  protected def updateUI(): Unit = { /* 修改GUI页面样式 */ }
}

object ButtonWithCallbacks {

  def apply(label: String, callback: () => Unit) =
    new ButtonWithCallbacks(label, List(callback))

  def apply(label: String) =
    new ButtonWithCallbacks(label, Nil)
}
```

`Widget` 是一个“标记”特征，我们之后会对该 trait 展开讨论。

```
// src/main/scala/progscala2/traits/ui/Widget.scala
package progscala2.traits.ui

abstract class Widget
```

点击按钮之后将触发一组类型为 `() => Unit`（这类函数完全通过副作用产生作用）的回调函数。

`ButtonWithCallbacks` 类有两大职责：更新可视界面样式（我们省略了相关代码）和处理回调行为。处理回调行为包括了管理一组回调函数以及点击按钮后调用这组函数。

我们在定义类型时应尽量做到职责分离，这样才能体现单一职责原则。单一职责原则认为每个类型都应该只做一件事，不应在一个类型中混杂多个职责。

我们希望能将按钮相关的逻辑从回调逻辑中抽离出来，这样一来这两类逻辑都会变得更加简单、更加模块化，也更易于测试和修改，可用性也更强。回调逻辑则很适用于实现成混入结构（mixin）。

我们将使用 trait 将回调逻辑从按钮逻辑中抽离出来。除此之外，我们会对逻辑抽离方式进行简单的概括。实际上，回调是观察者设计模式的一类特殊应用。因此，我们创建了两个 trait，分别用于声明观察者模式中的主体（Subject）和观察者（Observer），这两个 trait 同时还实现了主体和观察者的部分功能。之后我们会运用这两个 trait 处理回调行为。为了简化起见，我们首先使用一个简单的回调方法，统计按钮的点击次数：

```
// src/main/scala/progscala2/traits/observer/Observer.scala
package progscala2.traits.observer

trait Observer[-State] {                                        // ❶
  def receiveUpdate(state: State): Unit
}

trait Subject[State] {                                          // ❷
  private var observers: List[Observer[State]] = Nil            // ❸

  def addObserver(observer:Observer[State]): Unit =             // ❹
    observers ::= observer                                      // ❺

  def notifyObservers(state: State): Unit =                     // ❻
    observers foreach (_.receiveUpdate(state))
}
```

❶ 那些希望能在状态发生变化时得到通知的客户将运用这一 trait。这些客户代码必须实现 `receiveUpdate` 方法。

❷ 那些需要向观察者发送通知的主体应使用该 trait。

❸ 一组待通知的观察者列表。由于该列表为可变列表，因此非线程安全。

❹ 用于添加观察者的方法。

❺ 该表达式的意思是，把 `observer` 对象安放到 `observers` 列表的最前面，并将新生成的列表赋给 `observers` 列表。

❻ 该方法会向观察者通知状态变更。

通常，将混入了 `Subject` 特征的类直接设置为状态类型参数是最便捷的做法。因此，一旦某一对象的 `notifyObservers` 方法被调用了，该实例直接将自己作为参数传入即可，例如：直接传入 `this` 对象。

需要注意的是，由于 `Observer` 特征既未实现它所声明的方法，也未定义其他任何成员，在字节码级别，该 trait 实际上与 Java 8 之前的接口是完全等同的。由于缺少方法实现体明显是一个抽象对象，因此我们不需要使用 `abstract` 关键字。不过在声明那些包含了未实现方法体的抽象类时，我们必须使用 `abstract` 关键字，例如：

```
trait PureAbstractTrait {
  def abstractMember(str: String): Int
}

abstract class AbstractClass {
  def concreteMember(str: String): Int = str.length
  def abstractMember(str: String): Int
}
```

包含抽象方法的 trait 并不需要声明为抽象对象，无需在 trait 关键字之前添加 abstract 关键字。但是，那些包含一个或多个未定义方法的类则必须声明为抽象类。

我们继续对该示例进行讲解。由于 observers 列表是可变对象，因此下面两个表达式是等价的。

```
observers ::= observer
observers = observer :: observers
```

现在，我们将定义一个简化的 Button 类：

```
// src/main/scala/progscala2/traits/ui/Button.scala
package progscala2.traits.ui

class Button(val label: String) extends Widget {

  def click(): Unit = updateUI()

  def updateUI(): Unit = { /* 本方法包含了GUI样式的修改逻辑 */ }
}
```

Button 类非常简单，它只负责一件事情——处理点击事件。

下面我们将使用 Subject 特征。ObservableButton 类是 Button 类的子类，我们在该类中混入了 Subject 特征：

```
// src/main/scala/progscala2/traits/ui/ObservableButton.scala
package progscala2.traits.ui
import progscala2.traits.observer._

class ObservableButton(name: String)                          // ❶
    extends Button(name) with Subject[Button] {               // ❷

  override def click(): Unit = {                              // ❸
    super.click()                                             // ❹
    notifyObservers(this)                                     // ❺
  }
}
```

❶ Button 类的子类，该类混入了“可被观察”特性。

❷ ObservableButton 类继承了 Button 类，并混入了 Subject 特征。该类使用 Button 类型作为 Subject 特征的类型参数，在 Subject 特征声明中，该类型参数名为 State。

❸ 为了能够通知观察者，我们需要覆写 click 方法。假如存在其他受状态变化影响的方法，我们同样需要覆写这些方法。

❹ 首先，调用父类的 click 方法执行正常的 GUI 更新逻辑。

❺ 通知观察者们，将 this 对象作为 State 传递给 notifyObservers 方法。在当前场景中，除了按钮点击事件之外，不会发生其他事件。

我们试着运行下列脚本：

```
// src/main/scala/progscala2/traits/ui/button-count-observer.sc
import progscala2.traits.ui._
import progscala2.traits.observer._

class ButtonCountObserver extends Observer[Button] {
  var count = 0
  def receiveUpdate(state: Button): Unit = count += 1
}

val button = new ObservableButton("Click Me!")
val bco1   = new ButtonCountObserver
val bco2   = new ButtonCountObserver

button addObserver bco1
button addObserver bco2

(1 to 5) foreach (_ => button.click())

assert(bco1.count == 5)
assert(bco2.count == 5)
```

该脚本声明了一个观察者类型 ButtonCountObserver，该类型负责统计点击的次数。之后我们创建了两个 ButtonCountObserver 实例以及一个 ObservableButton 对象，这两个实例都被注册为按钮的观察者。之后该脚本点击了五次按钮，并使用 Predef 中定义的 assert 方法验证每个观察者统计的数量是否为 5。

假设我们只需要一个 ObservableButton 实例，那么我们并不需要单独声明一个混入了我们所需 trait 的类，只需要声明一个 Button 对象，并"凭空"为这个对象注入 Subject 特征即可：

```
// src/main/scala/progscala2/traits/ui/button-count-observer2.sc
import progscala2.traits.ui._
import progscala2.traits.observer._

val button = new Button("Click Me!") with Subject[Button] {

  override def click(): Unit = {
    super.click()
    notifyObservers(this)
  }
}

class ButtonCountObserver extends Observer[Button] {
  var count = 0
  def receiveUpdate(state: Button): Unit = count += 1
}

val bco1   = new ButtonCountObserver
val bco2   = new ButtonCountObserver

button addObserver bco1
button addObserver bco2
```

```
(1 to 5) foreach (_ => button.click())

assert(bco1.count == 5)
assert(bco2.count == 5)
println("Success!")
```

与之前的实现不同，我们在声明 Button() 对象时未声明新类，而是直接捎带上了 Subject[Button] 实例。这与 Java 中初始化某一实现了接口的匿名类的做法较为相似，但 Scala 提供了更多的灵活性。

> 如果待声明的类未扩展其他类，而只是混入了一些 trait，那么无论如何你必须使用 extend 关键字，并将其用于第一个 trait，而在其他的 trait 之前使用 with 关键字。但是，如果你想在实例化某一类型时混入 trait 声明，请在所有的 trait 之前添加 with 关键字。

在 8.8 节中，我推荐了一些适用于子类化（subclassing）的苛刻规则。接下来我们查看它们是否违背了这些“规则”。

- *一个抽象的基类或 trait，只被下一层的具体的类继承，包括 case 类*
  在这个例子中，我们并未使用抽象基类。
- *具体类的子类永远不会再次被继承，除了两种情况：(1) 类中混入了定义于 trait 中的其他行为*
  尽管 Button 类及它的子类 ObservableButton 类都是具体类，但后者混入了 trait 中的某些行为。
- *具体类的子类永远不会再次被继承，除了两种情况：(2) 只用于支持自动化单元测试的类*
  此处不适合该规则。
- *当使用子类继承可能是正确的做法时，考虑将行为分离到 trait 中，然后在类里混入这些 trait*
  我们正是这样做的！！
- *切勿将逻辑状态跨越父类和子类*
  按钮或其他 UI 小部件中的逻辑可视状态与通知相关的内部机制没有任何交集。因此我们仍然将 UI 状态封装在 Button 对象中，与观察者模式相关的状态则封装在 State 特征中。

## 9.3 可堆叠的特征

我们可以通过进一步地改进方案来提高代码的可重用性，使代码能同时使用多个 trait，例如“堆积”特征。

“可点击”性并非仅限于图形用户界面中的按钮。因此我们需要抽象出“可点击”这一逻辑。我们可以把该逻辑放到 Button 的父类（目前还只是个空类型）Widget 类中。不过并

不是所有的 GUI 小部件都需要接受点击事件，为此我们引入了另外一个 trait Clickable。

```
// src/main/scala/progscala2/traits/ui2/Clickable.scala
package progscala2.traits.ui2                                      // ❶

trait Clickable {
  def click(): Unit = updateUI()                                   // ❷

  protected def updateUI(): Unit                                   // ❸
}
```

❶ 我们在 traits.ui 包中已经实现了这些类型，所以此处使用了一个新的包名。

❷ click 方法是 public 方法，此处定义了该方法的具体实现，updateUI 方法将代理该方法。

❸ updateUI 方法既是 protected 方法，也是抽象方法。Clickable 特征的实现类将负责提供适当的实现逻辑。

由于 Button 示例中使用某一 protected 方法会对点击事件进行代理，所以我们无需为该示例引入 Clickable 这个简单接口。不过考虑到将公共抽象从具体实现细节中分离出来是一个很好的惯用方法，我们还是将它们融合到了一起。这也是“四人组”设计模式中所描述的*模版方法模式*（template method pattern）的一个简单示例。

下面列出了使用 Clickable 特征重构后的按钮定义：

```
// src/main/scala/progscala2/traits/ui2/Button.scala
package progscala2.traits.ui2
import progscala2.traits.ui.Widget

class Button(val label: String) extends Widget with Clickable {

  protected def updateUI(): Unit = { /* 修改GUI样式的逻辑代码 */ }
}
```

现在，Observation 特征应该依附于 Clickable 特征，而不再是之前的 Button 类。假如按照这种方式对代码进行重构，我们就不需要再关心如何观察按钮事件。但是由于我们确实需要观察像点击这样的事件，为此我们引入了仅用于观察 Clickable 事件的 trait：

```
// src/main/scala/progscala2/traits/ui2/ObservableClicks.scala
package progscala2.traits.ui2
import progscala2.traits.observer._

trait ObservableClicks extends Clickable with Subject[Clickable] {
  abstract override def click(): Unit = {        // ❶
    super.click()
    notifyObservers(this)
  }
}
```

❶ 请留意 abstract override 这些关键字，我们稍后会讨论它们。

该实现与之前的 ObservableButton 示例的实现非常相似。两者的关键区别点在于 abstract 关键字。在之前的示例中，我们只使用了 override 关键字。

我们再仔细观察一下 click 方法，该方法调用了 super.click() 方法。但是 super 指的是什么对象呢？在这个示例中，super 只能指向 Clickable 特征或 Subject 特征。Clickable 特征只是声明了 Click 方法，并未定义 Click 方法。而 Subject 特征则根本没有声明该方法。因此，super 并未绑定某一真实示例，至少当前未绑定。这也是为什么需要使用 abstract 关键字的原因。

事实上，当我们将该 trait 混入到像 Button 这种定义了 Click 方法的具体实例中时，super 便绑定了该实例。abstract 关键字提醒编译器（和读者）：尽管 ObservableClicks.click 方法提供了方法体，但 click 方法并没有完全实现。

> 只有在满足下面条件的情况下，我们才应在 trait 中定义的某一方法之前添加 abstract 关键字：该方法调用了 super 对象中的另一个方法，但是被调用的这个方法在该 trait 的父类中尚未定义具体的实现方法。

下面，我们将联合 ObservableClicks 特征，Button 对象及定义在 Button 类中的具体的 click 方法一起工作：

```
// src/main/scala/progscala2/traits/ui2/click-count-observer.sc
import progscala2.traits.ui2._
import progscala2.traits.observer._

// 无需覆写Button对象的click方法。
val button = new Button("Click Me!") with ObservableClicks

class ClickCountObserver extends Observer[Clickable] {
  var count = 0
  def receiveUpdate(state: Clickable): Unit = count += 1
}

val bco1 = new ClickCountObserver
val bco2 = new ClickCountObserver

button addObserver bco1
button addObserver bco2

(1 to 5) foreach (_ => button.click())

assert(bco1.count == 5, s"bco1.count ${bco1.count} != 5")
assert(bco2.count == 5, s"bco2.count ${bco2.count} != 5")
println("Success!")
```

请注意，假如希望观察点击事件，我们无需覆写 click 方法，直接声明一个 Button 实例并混入 ObservableClicks 特征便可。不仅如此，即便我们不希望观察点击事件，通过混入 Clickable 特征，我们还是可以构造出只支持点击操作的 GUI 对象。

尽管这种通过混入 trait 实现细粒度组合的功能非常强大，不过它有可能会被过度使用。

- 使用过多的 trait 会降低编译速度，这是因为编译器需要做一些额外的工作来合成输出的字节码。

- 类库用户在阅读代码或 Scaladoc 时，会惊讶地发现代码和文档中存在一长串的 trait 列表。

最后，在示例即将结束之际，我们再添加一个 trait。“JavaBean 规范”（JavaBean specification）提出了“可否决”事件（vetoable event）的思想。这一思想使得 JavaBean 状态变化的监听者能否决这一变更。我们可以应用 trait 实现相类似的功能：“否决”连续点击事件。你可以把该功能想象成阻止用户连续误点某一按钮而触发某一金融交易。下面列出了我们所实现的 VetoableClick 特征：

```
// src/main/scala/progscala2/traits/ui2/VetoableClicks.scala
package progscala2.traits.ui2
import progscala2.traits.observer._

trait VetoableClicks extends Clickable {                              // ❶
  // 默认的允许点击数。
  val maxAllowed = 1                                                  // ❷
  private var count = 0

  abstract override def click() = {
    if (count < maxAllowed) {                                         // ❸
      count += 1
      super.click()
    }
  }
}
```

❶ 该 trait 同样继承了 Clickable 特征。

❷ 允许点击的最大数量。（如果能提供“重置”机制，那就对用户更有帮助了。）

❸ 一旦点击数量超过了允许值（从 0 开始计数），后续点击事件便不会传递给 super 对象。

请注意 maxAllowed 被声明成了 val 数值，且注释进一步说明了 maxAllowed 的当前“默认”值，这表明 maxAllowed 的值是可以被修改的。那么如何修改这个 val 值呢？我们可以在混入了该 trait 的类或其他 trait 中覆写该值。在下面示例中，我们在使用了 ObservableClicks 和 VetoableClicks 特征的对象中将该值重新设置为 2。

```
// src/main/scala/progscala2/traits/ui2/vetoable-click-count-observer.sc
import progscala2.traits.ui2._
import progscala2.traits.observer._

// No override of "click" in Button required.
val button =
    new Button("Click Me!") with ObservableClicks with VetoableClicks {
  override val maxAllowed = 2                                     // ❶
}

class ClickCountObserver extends Observer[Clickable] {            // ❷
  var count = 0
  def receiveUpdate(state: Clickable): Unit = count += 1
}

val bco1 = new ClickCountObserver
val bco2 = new ClickCountObserver
```

```
button addObserver bco1
button addObserver bco2

(1 to 5) foreach (_ => button.click())

assert(bco1.count == 2, s"bco1.count ${bco1.count} != 2")       // ❸
assert(bco2.count == 2, s"bco2.count ${bco2.count} != 2")
println("Success!")
```

❶ 覆写 `maxAllowed` 的值，将该值设置为 2。

❷ 和之前一样，此处使用了相同的 `ClickObserver` 对象。

❸ 请注意，尽管按钮的实际点击数为 5，但此处预计显示的点击数现在是 2。

试着进行下面这个实验，切换一下声明 `button` 变量时指定 trait 的顺序：

```
val button = new Button("Click Me!") with VetoableClicks with ObservableClicks
```

执行这段代码后会发生什么呢？

这段代码无法通过断言，此时观察到的点击数是 5，而不是我们所期望的 2。

就像洋葱一样，我们对 `click` 方法同样进行了层层封装并实现了三种实现方式。那么问题来了，假如我们同时混入了 `VetoableClicks` 和 `ObservableClicks` 特征，系统会首先调用这三个方法中的哪个呢？声明顺序决定调用的顺序，系统将按照从右到左的顺序调用这些方法。

下面的伪代码解释了每个例子里的方法调用结构：

```
// new Button("Click Me!") with VetoableClicks with ObservableClicks

def ObservableClicks.click() = {
  if (count < maxAllowed) {       // super.click => VetoableClicks.click
    count += 1
    {
      updateUI()                  // super.click => Clickable.click
    }
  }
  notifyObservers(this)
}

// new Button("Click Me!") with ObservableClicks with VetoableClicks

def VetoableClicks.click() = {
  if (count < maxAllowed) {
    count += 1
    {                             // super.click => ObservableClicks.click
      {
        updateUI()                // super.click => Clickable.click
      }
      notifyObservers(this)
    }
  }
}
```

在第一个例子中，由于我们最后才声明 ObservableClicks 特征，因此 Clickable.click 方法仍然是可否决的，但观察者能知晓所有点击事件的发生。假如观察者希望只收到未否决点击事件的通知，你可以将该行为视为程序错误。

在第二个例子中，VetoableClick 特征位于声明序列的最后端，由于否决事件的逻辑封装了其他所有的 click 方法，因此只有当点击事件未被否决时，updateUI 和 notifyObservers 方法才会被调用。

Scala 使用线性化（linearization）算法解决具体类继承树中 trait 和类的优先级问题。这两类的按钮实现都很直观。trait 优先级遵循从右到左的原则，而按钮本身的代码体，例如：覆写 maxAllowed 值的代码会最后被估值。

我们会在 11.7 节详细地讨论线性化算法如何处理其他更加复杂的声明。

## 9.4　构造trait

尽管 trait 定义体起到了主构造函数的作用，不过 trait 主构造函数并不允许为其提供参数列表，而你也无法为其定义辅助构造函数。

在上一节的示例中，我们发现 trait 是可以继承其他的特征的。trait 同样也可以继承自类。不过，trait 无法向其父类构造函数传递参数，字面量参数也不例外。因此，trait 只能扩展自那些包含了无参主构造函数或无参辅助构造函数的类。

不过，就像类一样，每次创建使用了 trait 的实例时，特征体都会被执行。下面的脚本证实了这一点：

```
// src/main/scala/progscala2/traits/trait-construction.sc

trait T1 {
  println(s"  in T1: x = $x")
  val x=1
  println(s"  in T1: x = $x")
}

trait T2 {
  println(s"  in T2: y = $y")
  val y="T2"
  println(s"  in T2: y = $y")
}

class Base12 {
  println(s"  in Base12: b = $b")
  val b="Base12"
  println(s"  in Base12: b = $b")
}

class C12 extends Base12 with T1 with T2 {
  println(s"  in C12: c = $c")
  val c="C12"
  println(s"  in C12: c = $c")
}
```

```
println(s"Creating C12:")
new C12
println(s"After Creating C12")
```

执行该脚本会输出下列内容：

```
Creating C12:
  in Base12: b = null
  in Base12: b = Base12
  in T1: x = 0
  in T1: x = 1
  in T2: y = null
  in T2: y = T2
  in C12: c = null
  in C12: c = C12
After Creating C12
```

我们在脚本中定义了 `C12` 类的基类 `Base12`。运行脚本时会首先执行 `Base12` 类，之后再执行 `T1` 和 `T2` 特征体（依声明顺序从左到右的执行 trait），最后才是 `C12` 类体。

`T1` 和 `T2` 中 `println` 语句的执行顺序看上去与之前 `Clickable` 示例中的顺序相反。而实际上，它们的顺序是一致的。我们在这个示例中对初始化顺序进行了追踪，发现它是按照从左到右的顺序执行初始化的。

如果我们在声明按钮时依次输入了 `with VetoableClicks with ObservableClicks` 语句，这意味着我们首先将 `click` 方法定义为了 `VetoableClicks.click` 方法，而该方法会调用此时尚未实现的 `super.click` 方法。紧接着 Scala 将执行 `ObservableClicks` 特征体。在执行过程中，`VetoableClicks.click` 方法会被新的 `click` 方法所覆写，不过由于 `ObservableClicks` 同样会调用 `super.click` 方法，该方法会调用 `VetoableClicks.click` 方法。最后，由于 `ObservableClick` 继承了 `Clickable` 特征，而该 trait 提供了具体的 `click` 方法，这样一来，`VetoableClicks.click` 方法调用 `super.click` 方法时便会调用该 `click` 方法了。

因此，尽管我们无法向 trait 传递构造参数，但我们可以在特征体中初始化字段、方法和类型。线性化算法则允许声明中后续列出的 trait 或类覆写这些定义。假如某个 trait 或抽象父类中存在某些抽象成员，后续出现的 trait 或类必须对这些成员进行定义。这是因为具体类中不允许出现任何抽象成员。

> 请不要在 trait 中声明那些无法在初始化时指定合适默认值的具体字段。如果需要声明这类字段，请使用抽象字段，或者将该 trait 改成含有构造函数的类。当然，对于那些不包含状态信息的 trait 而言，初始化时不会遇到这些问题。

## 9.5　选择类还是trait

对于某些“概念”而言，应该使用 trait 来表示，还是使用类来表示呢？考虑这一问题时，请牢记一点：trait 是 Scala 实现混入的方法，它适用于大多数的“辅助”行为。假如

你发现某一特定的 trait 在大多数时候都被用作其他类的父类，那么这些子类表现得就像（behave as）这个父特征一样。为了使逻辑关系更加清晰，此时你应该考虑是否要把这个 trait 修改为类。（在这里，我们并未使用 is a 来表示子类与父特征的关系，而是用了 behave as 一词。这是因为根据里氏替换原则，is a 更适用于表示类的继承关系定义。）

良好的面向对象设计需要遵循下列的通用原则：一旦完成构造过程，该实例便应一直处于某种已知的合法状态中。

## 9.6 本章回顾与下一章提要

在本章中，我们学习了如何使用 trait 封装类及通过 trait 如何共享类之间的*横切关注点*（cross-cutting cocern）。我们也谈到了使用 trait 的时机和方法，并讲解了如何“叠加”多个 trait 以及初始化 trait 内值的相关规则。

在后面几章中，我们将深入学习 Scala 对象系统和类层次结构，会特别关注容器对象。

# 第10章

# Scala对象系统（I）

我们对 Scala 面向对象的实现已经有了一定的掌握。在本章中，我们将讨论标准库类型层次结构的更多详细信息，并深入探索其中的一些类型，如 `Predef`。

但是，在讨论标准库类型之前，我们首先来讨论一下被称为*继承转化*（variance under inheritance）的重要特性。在本章的最后，我们将讨论对象相等。

## 10.1 参数化类型：继承转化

Scala 参数化类型和 Java 参数化类型（在 Java 中，通常称为*泛型*，generic）的一个重要区别在于，继承差异机制如何工作。

假设一个方法带有的参数类型为 `List[AnyRef]`，你可以传入 `List[String]` 吗？换句话说，`List[String]` 是否应该被看作是 `List[AnyRef]` 的一个子类型呢？如果是，这种转化称为*协变*（covariance）。因为容器（被参数化的类型）的继承关系与参数类型的继承关系的“方向一致”。

同样存在类型是*逆变*（contravariant）的，对于特定类型 `X`，`X[String]` 是 `X[Any]` 的父类。

如果参数化类型既不是协变的，也不是逆变的，我们称之为*非转化*（invariant）的。相反地，有的参数化类型可以同时拥有两种或两种以上的这类属性。

Java 和 Scala 均支持协变，逆变和非转化类型。然而，在 Scala 中，转化行为的定义是类型声明的一部分，称为*转化标记*（variance annotation）。我们使用 `+` 来表示协变类型；使用 `-` 表示逆变类型；非转化类型不需要添加标记。换句话说，类型的设计者决定该类型在继承体系中如何进行转化。

以下是几个声明示例（很快我们就会接触真正的类型定义）：

```
class W[+A] {...}        // 协变
class X[-A] {...}        // 逆变
class Y[A] {...}         // 非转化
class Z[-A,B,+C] {...}   // 混合
```

相反，Java 中参数化的类型在定义时并未声明继承转化行为，而是在使用该类型时，也就是在声明变量时，才指定参数化类型的转化行为。

表 10-1 总结了 Java 和 Scala 中的三种转化标记及其意义。其中 Tsup 是 T 的父类，而 Tsub 是 T 的子类。

表10-1：类型的转化标记及其意义

| Scala | Java | 描　述 |
|---|---|---|
| +T | ? extends T | 协变（如 List[$T_{sub}$] 是 List[T] 的子类） |
| -T | ? super T | 逆变（如 X[$T^{sup}$] 是 X[T] 的子类） |
| T | T | 非转化继承（不能用 Y[$T^{sup}$] 或 Y[$T_{sub}$] 代替 Y[T]） |

回到 List 的讨论，Scala 的 List 实际上被声明为 List[+A]，意味着 List[String] 是 List[AnyRef] 的子类，所以对于类型参数 A，List 是协变的。当 List 只有一个协变的类型参数时，你会经常听到一种简称，即“列表是协变的”。相应地，对于逆变类型也有类似的叫法。

协变和非转化类型是容易理解的，那么逆变类型呢？

## 10.1.1　Hood下的函数

逆变的最好例子是一组 trait FunctionN，例如：scala.Function2（http://www.scala-lang.org/api/current/scala/Function2.html），其中 N 是一个介于 0 和 22（含 22）之间的数字，并且与函数所带的参数个数相对应。Scala 使用这些 trait 来实现匿名函数。

我们在本书中一直在使用匿名函数，匿名函数也称为函数字面量。例如：

```
List(1, 2, 3, 4) map (i => i + 3)
// 结果: List(4, 5, 6, 7)
```

函数表达式 i => i + 3 实际上是一个语法糖，编译器将其转化为 scala.Function1（http://www.scala-lang.org/api/current/scala/Function1.html）的匿名子类，其实现如下所示：

```
val f: Int => Int = new Function1[Int,Int] {
  def apply(i: Int): Int = i + 3
}
List(1, 2, 3, 4) map (f)
// 结果: List(4, 5, 6, 7)
```

当对象后面跟上参数列表时，就会调用默认的 apply 函数。这一约定源于函数应用（function application）的思想。例如：一旦定义了 f，我们就通过指定参数列表的方式调用它，如 f(1)。实际上 f(1) 是 f.apply(1)。

从历史上看，JVM 不允许字节码中出现“裸”函数。一切都必须包装在对象中。Java 的最近版本，特别是 Java 8，放宽了这种限制，但为了使 Scala 还能在旧版本的 JVM 中工作，编译器会将匿名函数转为 `FunctionN` 特征的匿名子类。在 Java 项目中，你或许已经为 Java 的接口写过很多这样的匿名子类了。

`FunctionN` 是抽象的，因为其中的 `apply` 方法是抽象方法。请注意，当我们使用更简洁的代码 `i => i +3` 时，编译器为我们定义了 `apply` 方法。匿名函数的函数体就是用来定义 `apply` 的。

> Java 8 增加了对函数字面量的支持，称为 Lambda 函数。但不同于 Scala 的实现方式，Java 采用了另一种实现方式，因为 Scala 还要支持旧版的 JVM。

再来讨论逆变，以下是 `scala.Function2` 的声明：

```
trait Function2[-T1, -T2, +R] extends AnyRef
```

最后一个类型参数 `+R` 是返回类型，它是协变的。开头的两个类型参数分别是函数的第一个和第二个参数，它们是逆变的。对于其他 `FunctionN` 特征，对应于函数参数的类型参数都是逆变的。

所以，函数在继承时都具有混合变异的行为。

这究竟是什么意思呢？下面的例子可以帮助我们理解变异行为：

```
// src/main/scala/progscala2/objectsystem/variance/func.scX

class CSuper                { def msuper() = println("CSuper") }    // ❶
class C      extends CSuper { def m()      = println("C") }
class CSub   extends C      { def msub()   = println("CSub") }

var f: C => C = (c: C)      => new C              // ❷
    f         = (c: CSuper) => new CSub           // ❸
    f         = (c: CSuper) => new C              // ❹
    f         = (c: C)      => new CSub           // ❺
    f         = (c: CSub)   => new CSuper         // ❻ 编译错误!
```

❶ 定义一个三层的类继承结构。

❷ 我们将函数 `f` 定义为 `var` 变量，完成一直对 `f` 赋值。所有有效的函数实例必须是 `C => C` 的形式（换句话说，就是 `Function1[C,C]` 的形式；同时也注意一下我们如何使用字面量语法）。我们用来赋值的变量还必须满足函数继承变异规则的约束。第一个赋值的变量是 `(c: C) => new C`（忽略了参数 `c`）。

❸ 该函数 `(c: CSuper) => new CSub` 是有效的，因此参数 `C` 是逆变的，可以用 `CSuper` 代替。如果返回值是协变的，`CSub` 可以代替 `C`。

❹ 类似❸，不过我们直接返回了 `C`。

❺ 类似❸和❹，不过我们直接传入了 `C`。

❻ 错误！函数携带 CSub 参数是无效的，因为参数是逆变的；返回值 CSuper 也是无效的，因为返回值是协变的。

这个脚本不产生任何输出。如果运行它，它会在最后一行编译失败，但其他语句还是有效的。

契约式设计（http://archive.eiffel.com/doc/manuals/technology/contract/）解释了为什么这些规则是有意义的。这是里氏替换原则的一种表现形式，我们会将它作为一个编程工具在 23.5 节进行简要讨论。现在，我们凭直觉尝试理解这些规则如此运行的原因。

函数变量 f 的类型是 C => C（that is，Function1[-C,+C]）。对 f 的第一次赋值与该签名完全相符。

现在，我们用不同的匿名函数对 f 赋值。添加的空格使得每次赋值与原始声明的相同点和差异变得更加明显。我们会不断地对 f 做重新赋值，直到测试出函数 C => C 的合法替换者。

第二次赋值，（x：CSuper）=> Csub，符合声明，即参数逆变，返回值协变。但这个声明为什么是安全的呢？

关键是了解 f 是如何调用的，以及我们可以对 f 背后的真正函数做哪些假设。当我们说 f 的类型是 C => C 时，我们其实定义了一个契约。这样，任何有效的 C 值都可以传给 f，f 也永远不会返回除 C 类值以外的任何值。

因此，如果实际的函数类型为（x:CSuper）=> Csub，该函数不仅可以接受任何 C 类值作为参数，也可以处理 C 的父类型的实例，或其父类型的其他子类型的实例（如果存在的话）。所以，由于只传入 C 的实例，我们永远不会传入超出 f 允许范围外的参数。从某种意义上说，f 比我们需要的更加“宽容”。

同样，当它只返回 Csub 时，这也是安全的。因为调用方可以处理 C 的实例，所以也一定可以处理 CSub 的实例。在这个意义上说，f 比我们需要的更加“严格”。

示例的最后一行同时打破了关于输入和输出类型的两个规则。如果允许这个函数合法地赋值给 f，我们考虑一下会发生什么。

在这种情况下，实际的 f 函数只知道如何处理 CSub 实例。但调用者对此一无所知，认为任何 C 实例都可以传入 f，所以当 f 运行出现“意外”时，就可能导致失败。即调用者试图把 C 实例传入到一个只接受 CSub 而不是 C 的函数。同样地，如果实际的 f 能够返回一个 CSuper 实例，这将超出调用者预期的返回值范围（预期返回 C 的实例）。

这解释了为什么函数的参数必须是逆变的，而返回值必须是协变的。

变异标记只有在类型声明中的类型参数里才有意义，对参数化的方法没有意义，因为该标记影响的是子类继承行为，而方法没有子类。例如 List.map 方法的简化签名：

```
sealed abstract class List[+A] ... {  // 忽略了混入的trait
  ...
  def map[B](f: A => B): List[B] = {...}
  ...
}
```

B 没有变异标记，如果你试图指定一个，编译器会抛出错误。

> 变异标记 +，表示参数化类型是协变的，- 标记表示参数化类型是逆变的。不带标记，表示该参数化类型是非变异的。

最后，编译器会检查你所用的编译标记是否有效。如果你试图在自定义的函数中加上错误的标记，以下就是发生的结果：

```
scala> trait MyFunction2[+T1, +T2, -R] {
     |   def apply(v1:T1, v2:T2): R = ???
     | }
<console>:37: error: contravariant type R occurs in covariant position
in type (v1: T1, v2: T2)R of method apply
         def apply(v1:T1, v2:T2): R = ???
             ^
<console>:37: error: covariant type T1 occurs in contravariant position
in type T1 of value v1
         def apply(v1:T1, v2:T2): R = ???
                   ^
<console>:37: error: covariant type T2 occurs in contravariant position
in type T2 of value v2
         def apply(v1:T1, v2:T2): R = ???
                          ^
```

注意以上的错误信息，编译器要求函数参数为逆变，返回值为协变。

### 10.1.2 可变类型的变异

到目前为止，我们讨论的参数化类型都是不可变类型。那么可变类型的变异行为是什么样的呢？答案是，可变类型只允许非变异行为。考虑以下示例：

```
// src/main/scala/progscala2/objectsystem/variance/mutable-type-variance.scX

scala> class ContainerPlus[+A](var value: A)
<console>:34: error: covariant type A occurs in contravariant position
in type A of value value_=
       class ContainerPlus[+A](var value: A)
             ^

scala> class ContainerMinus[-A](var value: A)
<console>:34: error: contravariant type A occurs in covariant position
in type => A of method value
       class ContainerMinus[-A](var value: A)
                                    ^
```

可变字段的问题在于，它像是一个私有的字段，却又存在公有的读写访问方法。即使可变字段是公有的并且没有显式定义的访问方法，该字段仍然会像私有字段一般运行。

回顾 8.6 节，`def value_=(newA: A)：Unit+` 是编译器为变量 `value` 生成的 setter 方法的签

名。也就是说，我们可以将表达式写为 myinstance.value = someA，然后调用该方法。需要注意的是，第一条错误信息给出了该方法的签名，并指出我们在应该使用逆变类型的位置使用了协变类型 A。

第二条错误信息给出了 => A 的方法签名。也就是说，该函数是一个不带参数，但返回类型为 A 的函数。这就像我们在 3.10 节第一次见到的按名调用参数。

以下声明是用显式方法声明来重写的，它看起来更像传统的 Java 代码：

```
class ContainerPlus[+A](var a: A) {
  private var _value: A = a
  def value_=(newA: A): Unit = _value = newA
  def value: A = _value
}
```

为什么传给 value_=(newA: A) 的 A 必须是逆变的呢？这看起来不对，因为我们要给 _value 赋一个新值，如果新值是 A 的父类实例，会出现一个类型错误。因为 _value 必须是类型 A，对吧？

其实，这种考虑思路是错误的。协变 / 逆变的规则适用于子类行为，而非超类行为。

假设以上的声明是有效的。我们可以用 C、CSub 和 CSup 实例化 ContainerPlus[C]：

```
val cp = new ContainerPlus(new C)   // ❶
cp.value = new C                    // ❷
cp.value = new CSub                 // ❸
cp.value = new CSuper               // ❹
```

❶ 在这里，类型参数 A 是类型 C。

❷ 有效：我们用的是与声明相同的类型实例。

❸ 按照通常的面向对象规则，这是有效的，因为 CSub 是 C 的子类。

❹ 编译错误。因为 CSup 的实例不能替换 C 的实例。

只有当考虑 ContainerPlus 的子类时，麻烦才出现：

```
val cp: ContainerPlus[C] = new ContainerPlus(new CSub)   // ❶
cp.value = new C             // ❷
cp.value = new CSub          // ❸
cp.value = new CSuper        // ❹
```

❶ 如果 ContainerPlus[+A] 有效，这一行就是有效的。

❷ 根据 c 的类型声明，这一行有效。这也是为什么参数类型必须逆变的原因。但该实例的 value_= 方法无法接受 C 的实例，因为该字段的类型是 CSub。

❸ 正确。

❹ 正确。

标❷的表达式说明了为什么方法的参数必须是逆变的。c 的用户期望它能与 C 的实例一起工作。通过观察 value_= 方法的实际实现，我们已经知道我们无法支持逆变，不过我们暂且忽略这一点，考虑如果修改变异标记会怎样：

```
class ContainerMinus[-A](var a: A) {
  private var _value: A = a
  def value_=(newA: A): Unit = _value = newA
  def value: A = _value
}
```

从本节开头的错误信息中，我们已经知道 value_= 方法被认为是正确的（尽管实际上该方法是不正确的），但我们现在获得了之前见过的第二条错误信息。value 方法的返回值类型为 A，所以 A 处于协变的位置。

为什么必须是协变的？这一点更直观一些。跟上次一样，子类的行为是关键。暂且假设编译器允许我们这样初始化 ContainerMinus 的实例：

```
val cm: ContainerMinus[C] = new ContainerMinus(new CSuper)   // ❶
val c: C      = cm.value        // ❷
val c: CSuper = cm.value        // ❸
val c: CSub   = cm.value        // ❹
```

❶ 如果 ContainerMinus[-A] 有效，这一行则是有效的。

❷ cm 认为其 value 方法返回值的类型为 C，但实际上该实例的 value 方法返回 CSuper 类型。

❸ 正确。

❹ 失败的原因同❷。

所以，对于 getter 和 setter 方法中的可变字段而言，它在读方法中处于协变的位置，而在写方法中又处于逆变的位置。不存在既协变又逆变的类型参数，所以对于可变字段 A 的唯一选择就是非变异。

## 10.1.3　Scala和Java中的变异

正如我们所说的，变异行为在 Scala 中定义于类型声明过程，而在 Java 中定义于使用过程。类型的客户端定义变异类型，设置默认类型为非变异。Java 不允许你在定义类型时指定变异行为，尽管你可以使用看起来类似的表达式。这些表达式定义了类型边界，我们将稍后进行讨论。

Java 指定变异行为时存在两个缺点。首先，库的设计者应该负责类型的编译行为并编写进库中。但现在是库的用户承担这个负担。这导致了第二个缺点，就是 Java 程序员很容易指定错误的变异注释，从而导致不安全的代码，就像我们刚刚讨论过的那样。

Java 类型系统的另一个问题是，Array 是协变的。考虑以下示例：

```
// src/main/java/progscala2/objectsystem/JavaArrays.java
package progscala2.objectsystem;

public class JavaArrays {
  public static void main(String[] args) {
    Integer[] array1 = new Integer[] {
      new Integer(1), new Integer(2), new Integer(3) };
    Number[] array2 = array1;      // 通过编译
```

```
    array2[2] = new Double(3.14);  // 通过编译,但会在运行时抛出错误!
  }
}
```

以上代码可以通过编译，但如果在 `sbt` 里运行，会抛出错误：

```
> run-main progscala2.objectsystem.JavaArrays
[info] Running progscala2.objectsystem.JavaArrays
[error] (run-main-4) java.lang.ArrayStoreException: java.lang.Double
java.lang.ArrayStoreException: java.lang.Double
  at progscala2.objectsystem.JavaArrays.main(JavaArrays.java:10)
  ...
```

哪里出错了？我们前面讨论过，可变集合的类型参数必须非变异，才是安全的。由于 Java 的数组是协变的，编译器允许我们对 `Array[Number]` 类型的引用赋一个 `Array[Integer]` 类型的值。然后编译器允许给这个数组中的元素赋以任何 `Number` 类型的值，但事实上，数组"知道"它只能接受 `Integer` 类型的值（如果有的话，也包括 `Integer` 的子类实例），所以它会抛出一个运行异常，破坏静态类型的检查机制。请注意，尽管 Scala 的数组是对 Java 数组的包装，但 `scala.Array`（http://www.scala-lang.org/api/current/#scala.Array）的类型参数是非变异的，所以它可以防止这个漏洞发生。

更多关于 Java 的泛型和数组的详细信息，请参阅 Maurice Naftalin 和 Philip Wadler 的 *Java Generics and Collections*，O'Reilly 出版社出版。本节最后一个例子就是改编自这本书。

## 10.2 Scala的类型层次结构

对 Scala 类型层次结构中的不少类型我们已经有了一定的了解。接下来，我们将了解该体系的一般结构，并掌握更多的详细信息。图 10-1 为顶层结构。除非另有说明，我们在这里讨论的所有类型都在 scala 的顶层包中。

`Any` 处于类型结构树的根部位置，`Any` 没有父类，但有三个子类：

- `AnyVal`，价值类型和价值类的父类。
- `AnyRef`，所有引用类型的父类。
- 通用特征（universal trait），新引入的 trait 类型，用于我们在 8.2 节讨论的特殊用途。

`AnyVal` 有九个具体子类，称为值类型。其中七个是数字值类型：`Byte`、`Char`、`Short`、`Int`、`Long`、`Float` 和 `Double`。余下的两个是非数字值类型，`Unit` 和 `Boolean`。

另外，正如 8.2 节讨论的那样，Scala 2.10 引入了用户自定义的值类，该值类继承自 `AnyVal`。

相反，所有其他类型均为引用类型。它们派生自 `AnyRef`，`AnyRef` 是 `java.lang.Object`（http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html）的别名。在 Java 的对象模型中，`Object` 并没有一个封装了原生类型和引用类型的父类，因为 Java 对原生类型做了特殊处理。

在 Scala 2.10 之前，编译器对所有 Scala 的引用类型混入了名为 `ScalaObject` 的 marker 特征。Scala 2.11 之后，编译器将不再这么做，该 trait 将会被移除。

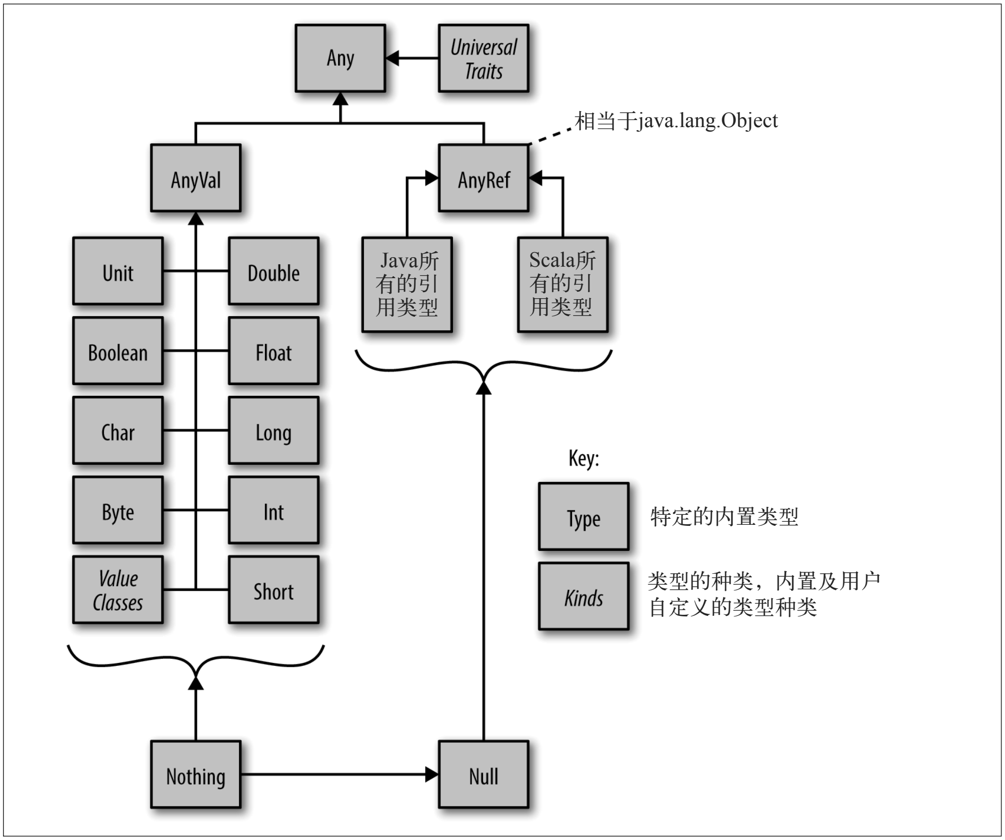



图 10-1：Scala 的类型层次结构

我们已经掌握了很多的引用类型，随着阅读的继续，我们将遇到更多。不过，这时我们会讨论一些被广泛使用的类型。

## 10.3 闲话Nothing（以及Null）

`Nothing`（http://www.scala-lang.org/api/current/#scala.runtime.Nothing$）和 `Null`（http://www.scala-lang.org/api/current/#scala.runtime.Null$）是位于类型系统底层的两个特殊类型。其中，`Nothing` 是所有其他类型的子类，而 `Null` 是所有引用类型的子类。

`Null` 对于大多数语言而言是个熟悉的概念。尽管这些语言通常并没有定义 `Null` 类型，仅仅定义了关键字 `null`，用于向引用变量赋值，表示该变量实际上没有值。`Null` 在编译器里的实现相当于以下声明：

```
package scala
abstract final class Null extends AnyRef
```

为何 Null 可以同时为 final 和 abstract 呢？该声明不允许子类派生以及创建实例，但运行时环境提供了一个实例，就是我们熟悉和喜爱的 null（这……这……）。

Null 被明确定义为 AnyRef 的一个子类型，但它也是所有 AnyRef 类型的子类型。这是类型系统允许你用 null 给任何引用类型赋值的正常做法。另一方面，因为 null 不是 AnyVal 的子类型，所以不可以用 null 给 Int 赋值。于是，Scala 的 null 与 Java 的 null 完全相同，因为它必须与 Java 的 null 共存于 JVM。否则，Scala 会破坏 null 的概念，引起很多潜在的 bug。

与此相反，Nothing 在 Java 中没有类似的概念，但它填补了存在于 Java 类型系统中的空白。Nothing 在编译器里的实现相当于以下声明：

```
package scala
abstract final class Nothing extends Any
```

Nothing 实际上继承了 Any。根据类型系统里的构造规则，Nothing 是所有其他类型的子类，无论是引用类型还是值类型。换句话说，Nothing 继承了所有一切（everything），这让它的名字听起来很奇怪。

不同于 Null，Nothing 没有实例。但 Nothing 为类型系统提供了两种功能，这有助于实现健壮且类型安全的设计。

我们结合熟悉的 List[\+A]（http://www.scala-lang.org/api/current/scala/collection/immutable/List.html）类来说明第一个功能。现在我们知道 List 中的 A 是协变的，所以 List[String] 是 List[Any] 的一个子类（因为 String 是 Any 的子类）。进而可以将 List[String] 的实例赋值给 List[Any] 类型的变量。

Scala 为空列表声明了一个专用的类型 Nil（http://www.scala-lang.org/api/current/scala/collection/immutable/Nil$.html）。在 Java 中，Nil 必须是像 List 那样的参数化类型，但是很不幸，根据定义 Nil 不包含任何元素，所以 Nil[String] 和 Nil[Any] 是不同的类型（实际上却并无区别）。

Scala 通过 Nothing 解决了这个问题。Nil 的声明如下：

```
package scala.collection.immutable
object Nil extends List[Nothing] with Product with Serializable
```

我们将在下一节讨论 Product。Serializable 是熟悉的 Java“标记”接口，用来表示对象可以用 Java 的内置机制序列化。

需要注意的是，Nil 是一个继承了 List[Nothing] 的对象，它只需要一个实例，因为它没有携带任何“状态”（元素）。由于 List 的类型参数是协变的，对于任意类型 A，Nil 是所有 List[A] 的子类型。所以，我们不需要分开 Nil[A] 实例，一个实例就够了。

Nothing 和 Null 被称为底部的类型，因为它们处于类型层次结构的底端。也因为如此，它们是所有（或者大多数）类型的子类。

Nothing 的其他用途是表示终止程序，如抛出一个异常。回顾我们在 8.9 节见过的 ??? 方

法。我们可以临时调用 ??? 方法来定义其他的方法，使得方法定义完整，并通过编译。但如果调用该方法，就会抛出异常。以下是 ??? 的定义：

```
package scala
object Predef {
  ...
  def ??? : Nothing = throw new NotImplementedError
  ...
}
```

由于 ???“返回”了 Nothing，所以它可以被任何其他函数调用，无论该函数的返回值是什么。以下是一个病态的例子：

```
scala> def m(l: List[Int]): List[Int] = l map (i => ???)
m: (l: List[Int])List[Int]

scala> m(List(1,2,3))
scala.NotImplementedError: an implementation is missing
  at scala.Predef$.$qmark$qmark$qmark(Predef.scala:252)
  ...
```

需要注意的是，尽管 ??? 返回 Nothing，我们仍然期望 m 返回 List[Int] 类型且函数能够通过编译器类型检查。

更为实际的是，??? 被已声明但尚未定义的方法调用：

```
/** @return (mean, standard_deviation) */
def mean_stdDev(data: Seq[Double]): (Double, Double) = ???
```

scala.sys 包（http://www.scala-lang.org/api/current/index.html#scala.sys.package）定义了一个 exit 方法，用于程序的正常退出，类似于 Java 中 System 包（http://docs.oracle.com/javase/8/docs/api/java/lang/System.html）的 exit 方法。不过，sys.exit 返回的是 Nothing。

这意味着方法可以声明为它返回了一个“正常”的类型，但在必要的时候也可以调用 sys.exit，并通过编译器类型检查。一个常见的例子是以下用于处理命令行参数的方法，该方法在提供的选项不正确时就退出：

```
// src/main/scala/progscala2/objectsystem/CommandArgs.scala
package progscala2.objectsystem

object CommandArgs {

  val help = """
  |usage: java ... objectsystem.CommandArgs arguments
  |where the allowed arguments are:
  |  -h | --help                Show help
  |  -i | --in | --input path    Path for input
  |  -o | --on | --output path   Path for input
  |""".stripMargin

  def quit(message: String, status: Int): Nothing = {                // ❶
    if (message.length > 0) println(message)
    println(help)
```

```
    sys.exit(status)
  }

  case class Args(inputPath: String, outputPath: String)            // ❷

  def parseArgs(args: Array[String]): Args = {
    def pa(args2: List[String], result: Args): Args = args2 match {  // ❸
      case Nil => result                                             // ❹
      case ("-h" | "--help") :: Nil => quit("", 0)                   // ❺
      case ("-i" | "--in" | "--input") :: path :: tail =>            // ❻
        pa(tail, result copy (inputPath = path))                     // ❼
      case ("-o" | "--out" | "--output") :: path :: tail =>          // ❽
        pa(tail, result copy (outputPath = path))
      case _ => quit(s"Unrecognized argument ${args2.head}", 1)      // ❾
    }
    val argz = pa(args.toList, Args("", ""))                         // ❿
    if (argz.inputPath == "" || argz.outputPath == "")               // ⓫
      quit("Must specify input and output paths.", 1)
    argz
  }

  def main(args: Array[String]) = {
    val argz = parseArgs(args)
    println(argz)
  }
}
```

❶ 打印可选的消息，然后打印帮助信息，以特定的错误码退出。遵循 Unix 的惯例，0 表示正常退出，非零值用来表示非正常终止。需要注意，`quit` 返回的是 `Nothing`。

❷ case 类，用来保存根据参数列表得到的设置信息。

❸ 嵌套的递归函数，用来处理参数列表。我们采用惯用的做法，传入一个 `Args` 实例，用来累加新的设置信息（通过复制一份）。

❹ 输入结束，于是返回累加得到的设置。

❺ 用户寻求帮助信息。

❻ 对于输入的参数，接受三种选项 `-i`、`--in` 或 `--input`，后面跟上路径参数。值得注意的是，如果用户没有提供路径（也没有其他参数），这个 case 语句就不会命中。

❼ 用 `tail` 和更新过的 `result` 调用 `pa`。

❽ 用 `--output` 参数重复 `--input` 的行为。

❾ 处理无法识别的参数引起的错误。

❿ 调用 `pa`，处理参数。

⓫ 确认参数中提供了输入参数或输出参数。

我发现本例中的类型匹配特别优雅而且简洁。

当你构建工程时，代码就被编译了。我们试着在 `sbt` 中运行该代码：

```
> run-main progscala2.objectsystem.CommandArgs -f
[info] Running progscala2.objectsystem.CommandArgs -f
```

```
Unrecognized argument -f

usage: java ... progscala2.objectsystem.CommandArgs arguments
where the allowed arguments are:
  -h | --help               Show help
  -i | --in | --input path    Path for input
  -o | --on | --output path   Path for input

Exception: sbt.TrapExitSecurityException thrown from the
  UncaughtExceptionHandler in thread "run-main-1"
  java.lang.RuntimeException: Nonzero exit code: 1
  at scala.sys.package$.error(package.scala:27)
[trace] Stack trace suppressed: run last compile:runMain for the full output.
[error] (compile:runMain) Nonzero exit code: 1
[error] ...

> run-main progscala2.objectsystem.CommandArgs -i foo -o bar
[info] Running progscala2.objectsystem.CommandArgs -i foo -o bar
Args(foo,bar)
[success] ...
```

对于无效的参数我们通常并不抛出异常，但 sbt 不喜欢我们调用 exit，于是抛出了异常。

## 10.4　Product、case类和元组

你定义的 case 类会混入 scala.Product 特征，它提供了几个关于实例字段的通用方法。例如，对于 Person 的实例：

```
scala> case class Person(name: String, age: Int)
defined class Person

scala> val p: Product = Person("Dean", 29)
p: Product = Person(Dean,29)  // case类实例分配到Product变量。

scala> p.productArity
res0: Int = 2                 // 字段的数量。

scala> p.productElement(0)
res1: Any = Dean              // 元素计算从0开始。

scala> p.productElement(1)
res1: Any = 29

scala> p.productIterator foreach println
Dean
29
```

尽管以通用方法访问字段非常有用，但由于对各个字段使用的是 Any 类型，而不是其具体类型，这种机制的作用受到了局限。

对于不同的字段数量，也有 Product 的子类型（例如：scala.Product2，http://www.scala-lang.org/api/current/scala/Product2.html，用于处理两个元素的场景），最大值为 22。这些类型为特定的字段添加了一些方法，可以保持该字段的正确类型信息。例如：

Product2[+T1,+T2] 增加了以下方法：

```
package scala
trait Product2[+T1, +T2] extends Product {
  abstract def _1: T1
  abstract def _2: T2
  ...
}
```

这些方法返回了字段的真正类型。这里的类型参数是协变的，因为 ProductN 特征只用于不可变的类型。用类似 _1 的方法访问这些字段需要使用对应的类型参数 T1，T1 处在协变的位置（即返回值类型）。

回顾一下，这些方法与用来访问元组元素的方法是相同的。事实上，所有的 TupleN 类型都继承了对应的 ProductN 特征，并提供了 _1 到 _N 方法的具体实现，N 最大可为 22：

```
scala> val t2 = ("Dean", 29)
t2: (String, Int) = (Dean,29)

scala> t2._1
res0: String = Dean

scala> t2._2
res2: Int = 29

scala> t2._3
<console>:36: error: value _3 is not a member of (String, Int)
              t2._3
                 ^
```

Tuple2 没有第三个元素，也没有 _3 方法。

为什么个数的上限是 22？这个数字的选择有些随意，但你可以合理地认为元组中有 22 个元素无论如何都已经足够多了。

这对于人类来说确实如此，但不幸的是，存在一个常见的情景需要超出这个数量限制：保存大的数据“记录”中的字段（或列）。对于 SQL 或 NoSQL 数据集，包含超过 22 个元素的情况并非罕见。元组很有用，至少对于小数据的确如此，因为元组可以保持字段（列）的顺序和类型。所以，22 个元素的限制是一个问题。

事实证明，在 Scala 2.10 中，case 类也受到不超过 22 个字段的限制。但这个实现上的限制在 2.11 版本中取消了，所以数据应用可以为超过 22 个元素的数据记录使用 case 类。

> 在 Scala 2.10 及更早的版本中，case 类被限制为拥有 22 个或更少的字段。这一限制在 Scala 2.11 中被取消了。

我们希望 Scala 在未来的版本中可以取消 22 个元素对 trait 和 Product 的限制。

## 10.5 Predef对象

为了方便起见，只要你编译代码，Scala 编译器就会自动导入顶层 Scala 包（名为 scala）以及在 java.lang 包（就像 javac 的）中的定义。因此，许多常见的 Java 和 Scala 类型都可以不经过明显地导入或提供完整的名称就可以使用。另外，编译器还导入了 Predef 对象中的一些定义，它提供了很多实用的定义，其中大部分的定义我们在之前已经讨论了。

接下来，我们来详细了解 Predef 提供的特性。需要注意的是，Scala 2.11 版本的 Predef 引入了很多变化，其中大部分是不可见的。

### 10.5.1 隐式转换

首先，Predef 定义了很多隐式转换规则，以下是一组转换对 AnyVal 类型的包装：

```
@inline implicit def byteWrapper(x: Byte)       = new runtime.RichByte(x)
@inline implicit def shortWrapper(x: Short)     = new runtime.RichShort(x)
@inline implicit def intWrapper(x: Int)         = new runtime.RichInt(x)
@inline implicit def charWrapper(c: Char)       = new runtime.RichChar(c)
@inline implicit def longWrapper(x: Long)       = new runtime.RichLong(x)
@inline implicit def floatWrapper(x: Float)     = new runtime.RichFloat(x)
@inline implicit def doubleWrapper(x: Double)   = new runtime.RichDouble(x)
@inline implicit def booleanWrapper(x: Boolean) = new runtime.RichBoolean(x)
```

Rich* 类型添加了额外的方法，类似于 <= 和 compare 等比较方法。@inline 标记鼓励编译器对它做内联，即直接将 new runtime.RichY(x) 逻辑插入到代码中。

举个例子，对于字节，为什么要有两个单独的类型？为什么不把所有的方法直接放在 Byte 中？原因是根据字节码的实现要求，额外的方法迫使程序在堆中分配一个实例。如果是像其他 AnyVal 类型一样的 Byte，则不分配在堆中，而是作为 Java 的原生类型。所以，拥有单独的 Rich* 类型是为了避免堆内存分配（除了有时需要使用那些方法的时候以外）。

在 scala.collection.mutable.WrappedArray（http://www.scala-lang.org/api/current/index.html#scala.collection.mutable.WrappedArray）中还存在用于包装 Java 的可变数组的方法，为数组添加了许多我们在第 6 章中讨论的集合方法：

```
implicit def wrapIntArray(xs: Array[Int]): WrappedArray[Int]
implicit def wrapDoubleArray(xs: Array[Double]): WrappedArray[Double]
implicit def wrapLongArray(xs: Array[Long]): WrappedArray[Long]
implicit def wrapFloatArray(xs: Array[Float]): WrappedArray[Float]
implicit def wrapCharArray(xs: Array[Char]): WrappedArray[Char]
implicit def wrapByteArray(xs: Array[Byte]): WrappedArray[Byte]
implicit def wrapShortArray(xs: Array[Short]): WrappedArray[Short]
implicit def wrapBooleanArray(xs: Array[Boolean]): WrappedArray[Boolean]
implicit def wrapUnitArray(xs: Array[Unit]): WrappedArray[Unit]
```

为什么要为每个 AnyVal 类型定义单独的方法？每个方法都用了自定义的 WrappedArray 的子类，表明 Java 原生类型的数组要比统一的数组更高效，因此就避免使用较为低效的、通用的引用类型的实现数组。

还有类似的方法，将数组转为 scala.collection.mutable.ArrayOps（http://www.scala-lang.

org/api/current/scala/collection/mutable/ArrayOps.html）。WrappedArray 与 ArrayOps 的唯一区别在于 WrappedArray 的转化函数（如 filter），会返回一个新的 WrappedArray，而对应的 WrappedOps 中的转化函数则返回 Array。

类似 WrappedArray 与 WrappedOps，String 也有相应的包装类型 scala/collection/immutable/WrappedString（http://www.scala-lang.org/api/current/scala/collection/immutable/WrappedString.html）和 scala/collection/immutable/StringOps（http://www.scala-lang.org/api/current/scala/collection/immutable/StringOps.html）。它们给 String 增加了集合方法，将其视为 Char 元素的集合。所以，Predef 定义了 String 和以上类型的相互转化：

```
implicit def wrapString(s: String): WrappedString
implicit def unwrapString(ws: WrappedString): String

implicit def augmentString(x: String): StringOps
implicit def unaugmentString(x: StringOps): String
```

有一对非常类似的包装类型，WrappedArray/ArrayOps 和 WrappedString/StringOps，它们看起来有些令人迷惑。不过，幸运的是，隐式转会自动触发，为需要的方法选择正确的包装类型。

还有很多其他方法可以实现 Java 包装的原生类型和 Scala 的 AnyVal 类型之间的转换。它们使得 Scala 和 Java 之间的互操作性更容易：

```
implicit def byte2Byte(x: Byte)           = java.lang.Byte.valueOf(x)
implicit def short2Short(x: Short)        = java.lang.Short.valueOf(x)
implicit def char2Character(x: Char)      = java.lang.Character.valueOf(x)
implicit def int2Integer(x: Int)          = java.lang.Integer.valueOf(x)
implicit def long2Long(x: Long)           = java.lang.Long.valueOf(x)
implicit def float2Float(x: Float)        = java.lang.Float.valueOf(x)
implicit def double2Double(x: Double)     = java.lang.Double.valueOf(x)
implicit def boolean2Boolean(x: Boolean)  = java.lang.Boolean.valueOf(x)

implicit def Byte2byte(x: java.lang.Byte): Byte             = x.byteValue
implicit def Short2short(x: java.lang.Short): Short         = x.shortValue
implicit def Character2char(x: java.lang.Character): Char   = x.charValue
implicit def Integer2int(x: java.lang.Integer): Int         = x.intValue
implicit def Long2long(x: java.lang.Long): Long             = x.longValue
implicit def Float2float(x: java.lang.Float): Float         = x.floatValue
implicit def Double2double(x: java.lang.Double): Double     = x.doubleValue
implicit def Boolean2boolean(x: java.lang.Boolean): Boolean = x.booleanValue
```

最后，Scala 2.10 中的一组隐式转换，可以防止将 null 用来赋值。我们只举一个 Byte 的例子：

```
implicit def Byte2byteNullConflict(x: Null): Byte = sys.error("value error")
```

这种机制会触发以下的错误信息：

```
scala> val b: Byte = null
<console>:23: error: type mismatch;
 found   : Null(null)
```

```
 required: Byte
Note that implicit conversions are not applicable because they are ambiguous:
 both method Byte2byteNullConflict in class LowPriorityImplicits of
 type (x: Null)Byte and method Byte2byte in object Predef of type (x: Byte)Byte
 are possible conversion functions from Null(null) to Byte
       val b: Byte = null
                     ^
```

上述代码运行得很好，但错误信息不够清晰。报错信息表明隐式转换存在歧义。虽然该歧义是库有意引入的，但实际上它应该直接告诉我们：不应该把 `null` 赋值给任何变量。

以下是 Scala 2.11 产生的错误信息：

```
scala> val b: Byte = null
<console>:7: error: an expression of type Null is ineligible for
  implicit conversion
       val b: Byte = null
                     ^
```

Scala 2.11 去掉了转换函数，并给出了更好更简洁的错误信息。

### 10.5.2　类型定义

`Predef` 定义了若干的类型及类型别名。

为了鼓励使用不可变集合，`Predef` 为最常用的不可变集合定义了别名：

```
type Map[A, +B]       = collection.immutable.Map[A, B]
type Set[A]           = collection.immutable.Set[A]
type Function[-A, +B] = Function1[A, B]
```

用于二元素和三元素元组的两个转换别名在 2.11 版本中被废弃，因为它们较少被使用，且没有足够的价值来证明存在的理由：

```
type Pair[+A, +B]       = Tuple2[A, B]
type Triple[+A, +B, +C] = Tuple3[A, B, C]
```

支持类型推断的其他一些 `Predef` 类型。

- `final class ArrowAssoc[A] extends AnyVal`

  用于实现 `a -> b` 形式的语法，该语法用来创建二元素的元素。我们在 5.3 节中讨论过这个用法。

- `sealed abstract class <:<[-From, +To] extends (From) => To with Serializ able`

  类型 `From` 是类型 `To` 的子类的证据。我们在 5.2.4 节中讨论过。

- `sealed abstract class =:=[-From, +To] extends (From) => To with Serializ able`

  类型 `From` 和类型 `To` 相等的证据。在 5.2.4 节中我们曾提到过。

- `type Manifest[T] = reflect.Manifest[T]`

  用于保持在 JVM 的类型擦除机制中丢失的类型信息。有个与之类似的类型 `OptManifest`。我们将在 24.2.2 节中讨论它们。

其他类型，如：`scala.collection.immutable.List`（http://www.scala-lang.org/api/current/scala/collection/immutable/List.html），可以用 `Predef` 中嵌入的导入来获得可见性。部分类型的伴随对象也是如此，如 `=:=`、`Map` 和 `Set`。

## 10.5.3 条件检查方法

有时你希望断言某条件为真，希望它“快速失败”（尤其在测试时）。`Predef` 定义了许多有助于达到这个目的的方法。

- `def assert(assertion: Boolean)`

  测试条件是否为真，如果不为真，抛出 `java.lang.AssertionError` 异常。

- `def assert(assertion: Boolean, message: => Any)`

  类似前面的 `assert`，但增加了一个可选参数，该参数将被转为错误信息字符串。

- `def assume(assertion: Boolean)`

  与 `assert` 相同，但其表示当一段代码块（如方法）正确时，条件才为真。

- `def assume(assertion: Boolean, message: => Any)`

  类似前面的 `assume`，但增加了一个可选参数，该参数将被转为错误信息字符串。

- `def require(requirement: Boolean)`

  与 `assume` 相同，但根据 Scaladoc，其含义是调用方是否满足某些条件，也可以表示某个实现不能得出特定的结果。

- `def require(requirement: Boolean, message: => Any)`

  类似前面的 `require`，但增加了一个可选参数，该参数将被转为错误信息字符串。

尽管没有明确地表明，但所有这些断言方法被加上了 `@elidable`（`ASSERTION`）标记。这个 `@elidable` 标记告诉编译器，除非相应标记（在这里，标记就是 `ASSERTION`）的参数大于编译时确定的阈值，否则对代码中的定义不产生字节码。例如：`scalac -Xelide-below 2000` 将阻止所有标记参数值低于 2000 的定义生成字节码。2000 刚好是 `elidable`（http://www.scala-lang.org/api/current/index.html#scala.annotation.elidable$）伴随对象中为 `ASSERTION` 定义的值。更多 `@elidable` 的信息，请参阅 Scaladoc 页面（http://www.scala-lang.org/api/current/scala/annotation/elidable.html）。

## 10.5.4 输入输出方法

我们很享受直接输写 `println("foo")` 的便利，不需要 Java 那样冗长的等效写法 `System.out.println（"foo"）`。`Predef` 为我们提供了四种将字符串打印到 `stdout` 的形式。

- `def print(x: Any): Unit`

  将 `x` 转为字符串，然后写到标准输出，结尾不会自动添加换行。

- `def printf(format: String, xs: Any*): Unit`

  用 `format` 和其他参数 `xs` 对 `printf` 风格的字符串进行格式化，然后将结果写到标准输出，结尾不会自动添加换行。

- `def println(x: Any): Unit`

  类似 `print`，但结尾会自动添加换行。

- `def println(): Unit`

  向标准输出打印空行。

Scala 2.10 中的 `Predef` 还为 `stdin` 中的读取输入定义了若干方法。然而，这些方法在 Scala 2.11 中被废弃了。在 Scala 2.11 中，它们被定义在新的 `scala.io.ReadStdin` 对象中。但是，这些方法的签名和行为是相同的。

- `def readBoolean(): Boolean`

  从标准输入的一个整行中读取一个 `Boolean` 值。

- `def readByte(): Byte`

  从标准输入的一个整行中读取一个 `Byte` 值。

- `def readChar(): Char`

  从标准输入的一个整行中读取一个 Char 值。

- `def readDouble(): Double`

  从标准输入的一个整行中读取一个 `Double` 值。

- `def readFloat(): Float`

  从标准输入的一个整行中读取一个 `Float` 值。

- `def readInt(): Int`

  从标准输入的一个整行中读取一个 `Int` 值。

- `def readLine(text: String, args: Any*): String`

  向标准输出中打印格式化的提示文本，并从标准输入中读取一整行字符串。

- `def readLine(): String`

  从标准输入中读取一整行字符串。

- `def readLong(): Long`

  从标准输入的一个整行中读取一个 `Long` 值。

- `def readShort(): Short`

  从标准输入的一个整行中读取一个 `Short` 值。

- `def readf(format: String): List[Any]`

  根据 `format` 中的区分符号，从标准输入中读取格式化输入。

- `def readf1(format: String): Any`

  根据 `format` 中的区分符号，从标准输入中读取格式化输入。并根据 `format` 的指定，返回提取的第一个值。

- `def readf2(format: String): (Any, Any)`

  根据 `format` 中的区分符号，从标准输入中读取格式化输入。并根据 `format` 的指定，返回提取的前两个值。

- `def readf3(format: String): (Any, Any, Any)`

  根据 `format` 中的区分符号，从标准输入中读取格式化输入。并根据 `format` 的指定，返回提取的前三个值。

> 你可以用 `scala.Console`（http://www.scala-lang.org/api/current/index.html#scala.Console$）中的方法覆写标准输入相关的 `java.io.Reader`（http://docs.oracle.com/javase/8/docs/api/java/io/Reader.html），或标准输出和标准错误相关的 `java.io.OutputStreams`（http://docs.oracle.com/javase/8/docs/api/java/io/OutputStream.html）。

### 10.5.5 杂项方法

最后，`Predef` 中还有一些更有用的方法需要突出介绍。

- `def ???: Nothing`

  在一个尚未实现的方法的方法体中调用。它为方法提供了具体的定义，允许编译器将方法所属的类型视为具体（与抽象对应）的类。然而，如果调用该方法，就会抛出 `scala.NotImplementedError`（http://www.scala-lang.org/api/current/scala/NotImplementedError.html）异常。我们在 8.9 节中第一次讨论了它。

- `def identity[A](x: A): A`

  直接返回参数 `x`。在将方法传给组合器（combinator）时，如果不需要进行修改，就可以用它。例如：在一个工作流程中，我们通过给 `map` 传入一个函数来对集合元素进行转化。有时我们不需要做任何转化，你可以传入 `identity`。

- `def implicitly[T](implicit e: T): T`

  当隐式参数列表使用简写 `[T:M]` 时，编译器会添加形式为 `(implicit arg: M[T])` 的隐式参数列表（实际名称不是 `arg`，而是编译器合成的唯一名称）。调用 `implicitly` 可以返回参数 `arg`。5.1 节中的“使用隐式”部分我们曾讨论过。

现在，让我们考虑一下面向对象程序设计中的一个重要主题，即如何检查对象是否相等。

## 10.6 对象的相等

很难准确地为实例实现一个可靠的相等测试。Joshua Block 的畅销书 *Effective Java*（Addison-Wesley 出版社出版）以及关于 `AnyRef.eq`（http://www.scala-lang.org/api/2.10.4/#scala.AnyRef）的 Scaladoc 页面都描述了相等测试需要满足的要求。

Martin Odersky、Lex Spoon 和 Bill Venners 共同撰写了一篇关于 `equals` 和 `hashCode` 方法的非常棒的文章“如何用 Java 语言编写相等方法”（“How to Write an Equality Method in

Java”，http://www.artima.com/lejava/articles/equality.html）。回想一下，在 case 类中，这些方法是自动创建的。

事实上，我从来没有写过我自己的 `equals` 和 `hashCode` 方法。我觉得，对于任何要使用的对象，如果可能需要测试相等性或需要作为 `Map` 的键（此时会调用 `hashCode`）的话，它们都应该定义为 case 类！



有的相等方法与其他语言中的相等方法名称相同，但语义有时是不同的！

接下来，我们学习用于测试相等的不同方法。

## 10.6.1　equals方法

我们将用一个 case 类来展示不同的相等方法是如何工作的：

```
// src/main/scala/progscala2/objectsystem/person-equality.sc

case class Person(firstName: String, lastName: String, age: Int)

val p1a = Person("Dean", "Wampler", 29)
val p1b = Person("Dean", "Wampler", 29)
val p2  = Person("Buck", "Trends",  30)
```

`equals` 方法用于测试值的相等，也就是说，如果 `obj1` 和 `obj2` 有相同的值，`obj1 equals obj2` 为 true。`obj1` 和 `obj2` 不需要指向同一个实例：

```
p1a equals p1a     // = true
p1a equals p1b     // = true
p1a equals p2      // = false
p1a equals null    // = false
null equals p1a    // 抛出 java.lang.NullPointerException异常
null equals null   // 抛出 java.lang.NullPointerException异常
```

所以，`equals` 的行为类似 Java 里的 `equals` 方法和 Ruby 里的 `eql?` 方法。

## 10.6.2　==和!=方法

`==` 在很多语言中是一个操作符，但在 Scala 中是一个方法。在 Scala 2.10 中，`==` 在 `Any` 中被定义为 `final`，用来代表 `equals`。在 2.11 版本中改变了实现，但行为保持不变：

```
p1a == p1a        // = true
p1a == p1b        // = true
p1a == p2         // = false
```

所以，`==` 的行为与 `equals` 完全一样，即只测试值是否相等。当 `null` 在 `==` 左边时是个例外：

```
p1a == null       // = false
```

```
null == p1a         // = false
null == null        // = true  (编译警告,这永远为true)
```

`null == null` 应该为 true 吗？实际上，这会产生一条警告：

```
<console>:8: warning: comparing values of types Null and Null using `=='
will always yield true
```

正如你预期的那样，`!=` 表示不相等，与 `!(obj1 == obj2)` 等价：

```
p1a != p1a          // = false
p1a != p1b          // = false
p1a != p2           // = true
p1a != null         // = true
null != p1a         // = true
null != null        // = false (编译警告,这永远为true)
```

在 Java、C++ 和 C# 中，`==` 操作符测试的是引用，而不是值。与此相反，Scala 的 `==` 测试的是值的相等性。

## 10.6.3 eq和ne方法

`eq` 方法用于测试引用的相等性。也就是说，如果 `obj1` 和 `obj2` 指向内存中的同一个位置，`obj1 eq obj2` 就为 true。这两个方法只对 `AnyRef` 类型有定义：

```
p1a eq p1a          // = true
p1a eq p1b          // = false
p1a eq p2           // = false
p1a eq null         // = false
null eq p1a         // = false
null eq null        // = true  (编译警告,这永远为true)
```

正如编译器为 `null == null` 提出警告，`null eq null` 也得到了相同的警告。

所以，`eq` 的行为就像 Java、C++ 和 C# 中的 `==` 操作符一样。

`ne` 方法与 `eq` 的相反，也就是说它与 `!(obj1 eq obj2)` 等价：

```
p1a ne p1a          // = false
p1a ne p1b          // = true
p1a ne p2           // = true
p1a ne null         // = true
null ne p1a         // = true
null ne null        // = false (编译警告,这永远为true)
```

## 10.6.4 数组相等和sameElements方法

比较两个数组，在 Scala 中并不能得出我们认为的显而易见的结果：

```
Array(1, 2) == Array(1, 2)              // = false
```

这令人惊讶。值得庆幸的是，有一个简单的解决方案，就是 sameElements 方法：

```
Array(1, 2) sameElements Array(1, 2)    // = true
```

实际上，我们最好要记住，Array 是我们熟知和喜爱的，它是可变的原始 Java 数组，与 Scala 库中我们习惯使用的集合类型有着不同的方法。

所以，如果你试图比较数组，考虑一下用序列来比较是否会更好。（不使用序列来代替的一个理由是，你有时真的需要数组相对于序列的优势性能。）

与数组相反，序列（比如 List）的相等性的行为就符合你的期望：

```
List(1, 2) == List(1, 2)              // = true
List(1, 2) sameElements List(1, 2)    // = true
```

## 10.7 本章回顾与下一章提要

我们讨论了 Scala 对象系统中的重要主题，如继承行为，Predef 的特性，类型层次结构的基础知识以及相等性。

接下来，我们将继续讨论对象系统，了解其中的成员覆写和解析规则。

# 第11章

# Scala对象系统（II）

本章将对类成员和trait成员的覆写规则进行介绍，并结束对Scala对象系统的讨论。具体内容包括：Scala如何通过*线性化算法*（linearization algorithm）解决成员定义以及如何对混入了trait并继承自其他类型的类型进行覆写。

## 11.1 覆写类成员和trait成员

我们可以在类中及trait中声明抽象成员，包括抽象字段、抽象方法和抽象类型。在创建实例前，继承类或trait必须定义这些抽象成员。大多数面向对象语言都支持抽象方法，其中某些语言还支持抽象字段和抽象类型。

> 假如你需要对Scala的某一具体成员进行覆写，覆写该成员时必须使用`override`关键字。假如某一子类型定义（也可以说“覆写”）了抽象成员，`override`关键字是可省略的。反过来说，如果并未覆写某一成员但却使用了`override`关键字，这会导致错误。

强制使用`override`关键字会带来下列好处。

- 有些成员本应对其他成员进行覆写，而使用`override`关键字能够捕获这些成员的拼写错误。假如这些成员未覆写任何成员，编译器便会抛出错误。
- 向基类中添加新的成员时，由于基类开发人员对继承类并不太了解，这就可能会出现新添的成员名与继承类中某一已经存在的成员名称冲突，而Scala能够捕获这一细微的错误。也就是说，我们不希望继承类中的成员对基类成员进行覆写，而由于该继承类成员未提供`override`关键字，当引入这个新的基类成员时，编译器便会抛出错误。
- 由于必须添加这一关键字，这有助于提醒你考虑到底哪些成员应该被覆写。

Java 提供了 `@Override` 对方法进行注解，你可以选择是否添加该注解。尽管该注解能够辅助用户捕获拼写错误，但是由于注解是可选的，因此它无法帮助用户捕获上面第二项所描述的细微错误。

在实现抽象方法时，能否选择性地使用 `override` 关键字呢？我们一起听听赞同和反对的声音。

除了之前提出的几点之外，赞同使用 `override` 关键字的理由还包括如下 2 点。

- 使用 `override` 关键字能提醒读者，定义在父类中的某一成员已经被实现了（也有可能被覆写了）。
- 假如父类移除了已经在某一子类中定义了的抽象成员，系统将会报错。

而反对使用 `override` 关键字的理由如下。

- 捕获拼写错误其实并没必要。未定义“覆写”也意味着当前仍然存在未定义的成员，因此继承类（或该类的其他具体类）无法通过编译。
- 在代码库的演变过程中，假如维护父类某一抽象成员的开发人员决定将该抽象成员改为具体成员，这一变化在编译子类时无法被注意到。现在子类应该调用该方法的父类实现吗？编译器会默默地使用子类方法覆写父类新定义的实现。

## 避免覆写具体成员

换句话说，抽象成员到底应不应该使用 `override` 关键字呢？我们很难给出答案。这一问题也引入了更多的问题：你是否应该覆写一个具体方法呢？正确的回答是，大多数情况下，你不应该这样做。

父类型和子类型的关系就好比一纸契约，我们需要费心确保子类没有破坏父类型所指定的实现行为。

覆写具体成员时，很容易破坏掉这纸契约。覆写 `foo` 方法时是否应该调用 `super.foo` 方法呢？如果调用了 `super.foo` 方法，子类的实现方法应该什么时候调用该方法呢？当然，正确的做法取决于具体的场景。

著名的“四人组”所编写的《设计模式》一书中描述的*模版方法模式*（template method pattern）构造了一个更为牢固的契约。在该模式中，父类提供了某一方法的具体实现，并以此定义了某一行为的主要轮廓。而需要使用多态行为时，该方法也会调用一些 `protected` 抽象方法。在此之后，子类型则只需要实现 `proteced` 抽象方法即可。

下面我们给出这样一个示例，说明供美国公司使用的工资计算器的粗略实现。

```
// src/main/scala/progscala2/objectsystem/overrides/payroll-template-method.sc

case class Address(city: String, state: String, zip: String)
case class Employee(name: String, salary: Double, address: Address)

abstract class Payroll {
  def netPay(employee: Employee): Double = {                    // ❶
    val fedTaxes   = calcFedTaxes(employee.salary)
    val stateTaxes = calcStateTaxes(employee.salary, employee.address)
```

```
    employee.salary - fedTaxes -stateTaxes
  }

  def calcFedTaxes(salary: Double): Double                        // ❷
  def calcStateTaxes(salary: Double, address: Address): Double      // ❸
}

object Payroll2014 extends Payroll {
  val stateRate = Map(
    "XX" -> 0.05,
    "YY" -> 0.03,
    "ZZ" -> 0.0)

  def calcFedTaxes(salary: Double): Double = salary * 0.25          // ❹
  def calcStateTaxes(salary: Double, address: Address): Double = {
    // Assume the address.state is valid; it's found in the map!
    salary * stateRate(address.state)
  }
}

val tom  = Employee("Tom Jones", 100000.0, Address("MyTown", "XX", "12345"))
val jane = Employee("Jane Doe",  110000.0, Address("BigCity", "YY", "67890"))

Payroll2014.netPay(tom)    // Result: 70000.0
Payroll2014.netPay(jane)   // Result: 79200.0
```

❶ netPay 方法应用了模版方法模式。该方法定义了计算工资的协议，并将像这类每年都会变化的具体细节委托给抽象方法处理。

❷ 计算美国联邦税。

❸ 计算州税。

❹ 定义父类中抽象方法的具体实现。

请注意，本实现中未出现 override 关键字。

这些天，当在代码中看到 override 关键字时，我便仿佛看到一个潜在的设计坏味（design smell）。某人也许正在对某一具体行为进行覆写，这可能会导致细微的 bug。

对于“不要覆写父类具体方法”这条规则，我能想到两个例外。某一方法的父类实现对于子类而言确实没有用处，这是其一。例如 toString、equals 和 hashCode 方法。不幸的是，覆写这些无用的默认方法非常普遍，以至于我们都满足于对具体方法进行覆写。

第二个例外则出现在这样一个场景（也许出现次数很少）：你需要混入某些非堆叠（non-overlapping）行为。例如：你也许会对某些重要方法进行覆写，以便调用日志接口。在子类覆写中，你调用日志方法时，使用 super 对象并不会影响该方法的外部行为（这也是该方法对外的契约）。但前提是你能正确调用父类方法！

> 除非具体方法是 toString 这样的无用方法，否则请不要覆写具体方法。除非你确实是在覆写具体方法，否则不要使用 override 关键字。

## 11.2　尝试覆写final声明

假如某一声明中包含 final 关键字，那么 Scala 不允许覆写该声明。在下面的示例中，fixedMethod 方法在父类中被声明为 final 方法。编译该示例时会出现编译错误：

```
// src/main/scala/progscala2/objectsystem/overrides/final-member.scalaX
package progscala2.objectsystem.overrides

class NotFixed {
  final def fixedMethod = "fixed"
}

class Changeable2 extends NotFixed {
  override def fixedMethod = "not fixed"   // 编译错误
}
```

final 除了用于对成员进行限制之外，还能用于限制类和 trait。在下面的示例中，由于 Fixed 类被声明为 final 类，因此当某一新类型试图继承 Fixed 类时，该新类型无法通过编译：

```
// src/main/scala/progscala2/objectsystem/overrides/final-class.scalaX
package progscala2.objectsystem.overrides

final class Fixed {
  def doSomething = "Fixed did something!"
}

class Changeable1 extends Fixed     // 编译错误
```

在 Scala 类库中，某些类型被声明为 final 类型。这些类型包括某些 JDK 类（如 String 类型），以及继承自 AnyVal 类型的所有“值”类型（请参考 10.2 节）。

## 11.3　覆写抽象方法和具体方法

对那些未声明为 final 类型的声明体，我们将对作用于这些声明体的覆写规则和行为进行考察。首先从方法开始。

我们之前在 9.2 节中引入了 Widget 特征，下面我们将对其进行扩展。为了使小部件（widget）能够在显示器或网页上展现，我们引入了抽象方法 draw。同时，像所有的 Java 程序员一样，我们还会对 toString() 这一具体方法进行覆写，将其转化成某一特定的字符串格式。

事实上，处理图形绘制时需要考虑多个相交的问题。Widget 对象的状态是问题之一；而如何将其渲染到不同的平台上则是另一个单独的问题。这些平台包括了“富”客户端、网页、移动设备等。因此，trait 非常适合处理绘图问题，尤其是当你希望 GUI 抽象体可移植的时候。不过为了简化问题，我们只会在 Widget 类层次结构中处理绘图逻辑。

下面列出了修订后的 Widget 类定义，新的定义中提供了 draw 方法和 toString 方法。从逻

辑上考虑，Widget 类是所有像按钮这样的 UI 小部件的父类，因此我们现在将 Widget 类变为抽象类。

```
// src/main/scala/progscala2/objectsystem/ui/Widget.scala
package progscala2.objectsystem.ui

abstract class Widget {
  def draw(): Unit
  override def toString() = "(widget)"
}
```

由于 draw 方法并未定义方法体，因此该方法是抽象方法，同理，Widget 类也需被声明成抽象类。Widget 类的所有具体子类均需要实现 draw 方法，当然，也可以依赖某一实现了该方法的父类。尽管 draw 方法可以返回某种“成功”状态，但我们并不需要 draw 方法返回事物，因此该方法返回值为 Unit。

toString() 方法的实现较为直观。由于 AnyRef 类定义了 toString 方法，因此声明 Widget.toString 方法时必须使用 override 关键字。

下面列出了修订后的 Button 类，该类同样实现了 draw 方法和 toString 方法：

```
// src/main/scala/progscala2/objectsystem/ui/Button.scala
package progscala2.objectsystem.ui
import progscala2.traits.ui2.Clickable

class Button(val label: String) extends Widget with Clickable {

  // Simple hack for demonstration purposes:
  def draw(): Unit = println(s"Drawing: $this")

  // From Clickable:
  protected def updateUI(): Unit = println(s"$this clicked; updating UI")

  override def toString() = s"(button: label=$label, ${super.toString()})"
}
```

Button 类也混入了我们在 9.3 节中引入的 Clickable 特征。我们稍后会进行深入说明。

我们可以将 Button 类实现为 case 类，不过正如你所见，之后将 Button 类进一步划分以生成其他的按钮类型。我们也希望避免之前讨论过的 case 类继承可能产生的问题。

Button 类实现了抽象方法 draw。而 override 关键字在此处是可选关键字。Button 类同时也覆写了 toString 方法，对 toString 方法的覆写则需要 override 关键字。请注意覆写后的 toString 方法调用了 super.toString 方法。

> 实现抽象方法时，是否应该使用 override 关键字呢？我认为不应使用。假设 Widget 类的维护人员将来决定提供默认的 draw 实现方法，实现该方法的目的也许是为了记录每次调用的情况。如果你已经对 draw 方法进行了覆写，编译器会默认你目前确实要对该具体方法进行覆写，而你永远不会知道父类的这个变更。不过，如果你未使用 override 关键字，那么当抽象方法 draw 突然有了实现体之后，你的代码会无法通过编译，因此你知道这个变更。

尽管 super 关键字与 this 作用类似，不过 super 绑定到父类型，即当前类的父类和其他混入的 trait。查找 super.toString 方法时会找到“最近”的父类型所提供的 toString 方法，而父类型的距离则是由本章稍后要讲到的线性化过程所决定的（请参考 11.7 节的相关内容）。在这个示例中，由于 Clickable 特征未定义 toString 方法，super.toString 将调用 Widget.toString 方法。我们在这个示例中延用了 9.3 节中引入的 Clickable 特征。

> 由于覆写具体方法容易产生错误，因此应尽量少用。那么我们是否应该调用父类方法呢？如果应该调用，什么时候调用比较合适呢？你是在执行其他业务逻辑之前调用父类方法，还是之后呢？尽管父类方法的作者也许会列出覆写该方法时的限制条件，不过很难确保继承类的作者会遵守这些限制。模版方法模式（template method pattern）是非常健壮的应用程序。

下面列举了一段简单的脚本，用于演示如何操作 Button 类：

```
// src/main/scala/progscala2/objectsystem/ui/button.sc
import progscala2.objectsystem.ui.Button

val b = new Button("Submit")
// Result: b: oop.ui.Button = (button: label=Submit, (widget))

b.draw()
// Result: Drawing: (button: label=Submit, (widget))
```

## 11.4 覆写抽象字段和具体字段

由于 trait 存在一些特殊的问题，我们将分开讨论 trait 和类的字段覆写。

### 覆写trait中的字段

下面列举了一段精心设计的示例代码。在字段初始化之前，该示例会调用这个尚未定义的字段：

```
// src/main/scala/progscala2/objectsystem/overrides/trait-invalid-init-val.sc
// ERROR: "value" read before initialized.

trait AbstractT2 {
  println("In AbstractT2:")
  val value: Int
  val inverse = 1.0/value      // ❶
  println("AbstractT2: value = "+value+", inverse = "+inverse)
}

val obj = new AbstractT2 {
  println("In obj:")
  val value = 10
}
println("obj.value = "+obj.value+", inverse = "+obj.inverse)
```

❶ 初始化 inverse 字段时，value 值是多少？

尽管看上去我们是通过 new AbstractT2 语句创建的 AbstractT2 特征的一个实例，不过事实

上我们实例化了一个隐式地扩展了 AbstractT2 特征的匿名类。请留意我们使用 scala 命令行（$ 是脚本提示符）运行该脚本时产生的输出信息：

```
$ scala src/main/scala/progscala2/objectsystem/overrides/trait-bad-init-val.sc
In AbstractT2:
AbstractT2: value = 0, inverse = Infinity
In obj:
obj.value = 10, inverse = Infinity
```

假如在 REPL 中执行 :load src/main/scala/progscala2/objectsystem/overrides/trait-bad-init-val.sc 命令，或是将代码复制到 REPL 中执行，你能得到执行的结果（与上面的输出相比，你会得到一些额外的输出行）。

正如你所预计的那样，inverse 变量过早被计算了。尽管没有抛出除零异常（divide-by-zero exception），但是编译器仍认为 inverse 值无穷大。

Scala 为此类问题提供了两个解决方案。第一个方案是使用惰性值（lazy value），我们之前在 3.11 节中曾对此进行了讨论：

```
// src/main/scala/progscala2/objectsystem/overrides/trait-lazy-init-val.sc

trait AbstractT2 {
  println("In AbstractT2:")
  val value: Int
  lazy val inverse = 1.0/value      // ❶
  // println("AbstractT2: value = "+value+", inverse = "+inverse)
}

val obj = new AbstractT2 {
  println("In obj:")
  val value = 10
}
println("obj.value = "+obj.value+", inverse = "+obj.inverse)
```

❶ 添加了 lazy 关键字，并注释了 println 语句。

现在，inverse 成功地被初始化，并被赋予了合法值：

```
In AbstractT2:
In obj:
obj.value = 10, inverse = 0.1
```

但是，只有当我们不使用 println 语句时，lazy 关键字才能起到作用。如果你移除了 // 符号后再执行该脚本，你会再次得到 Infinity 值。这是因为 lazy 会推迟对变量进行估值，直到有代码需要使用该值。而 println 语句过早要求 scala 对 inverse 变量进行估值。

> 假如某一 val 变量是惰性值，请确保尽可能地推迟对该 val 值的使用。

预先初始化字段是第二个解决方案，该方案并未得到广泛使用。请思考下面改良后的实现：

```
// src/main/scala/progscala2/objectsystem/overrides/trait-pre-init-val.sc

trait AbstractT2 {
  println("In AbstractT2:")
  val value: Int
  val inverse = 1.0/value
  println("AbstractT2: value = "+value+", inverse = "+inverse)
}

val obj = new {
  // println("In obj:")      // ❶
  val value = 10
} with AbstractT2

println("obj.value = "+obj.value+", inverse = "+obj.inverse)
```

❶ 预先初始化块中只允许出现类型定义或具体字段定义。比方说，如果此处调用了 `println` 语句，那么会出现编译错误。

在 `with AbstractT2` 子句执行之前，我们便已实例化了一个匿名内部类并在代码块中初始化了该内部类的值字段。这确保了在执行 `AbstractT2` 特征体之前，`value` 字段已初始化完毕。这点在我们执行脚本时便能发现：

```
In AbstractT2:
AbstractT2: value = 10, inverse = 0.1
obj.value = 10, inverse = 0.1
```

即便是在 `AbstractT2` 特征的主体中，`inverse` 也得到了很好的初始化。

现在我们将对 `VetoableClick` 特征进行覆写。我们之前在 9.3 节中使用过该 trait，它定义了一个名为 `maxAllowed` 的 `val` 变量，并将其初始化为 `1`。我们希望能够在某个混入了该 trait 的类中对该值进行覆写。下面再次列出了 `VetoableClicks` 的实现代码：

```
// src/main/scala/progscala2/traits/ui2/VetoableClicks.scala
package progscala2.traits.ui2
import progscala2.traits.observer._

trait VetoableClicks extends Clickable {                    // ❶
  // 默认的允许点击数
  val maxAllowed = 1                                        // ❷
  private var count = 0

  abstract override def click() = {
    if (count < maxAllowed) {                               // ❸
      count += 1
      super.click()
    }
  }
}
```

❶ `VetoableClicks` 依然扩展自 `Clickable` 特征。

❷ 最大允许点击数。（如果能提供“重置”功能，这对该 trait 将会有所帮助。）

❸ 点击数一旦超过了允许值（从 0 开始计数），后续的点击事件便不会发送给 `super` 对象。

尽管你可以很直观地创建混入了该 trait 的实例并按要求对 maxAllowed 变量进行覆写，但是我们首先应该审视一下，初始化这样一类实例时会出现哪些问题。

为了能发现这些问题，我们首先回到 VetoableClicks 特征，并将该 trait 混入 Button 类。为了能够检测到发生了什么，我们同样也需要混入之前在 9.3 节中讨论过的 ObservableClicks 特征：

```
// src/main/scala/progscala2/traits/ui2/ObservableClicks.scala
package progscala2.traits.ui2
import progscala2.traits.observer._

trait ObservableClicks extends Clickable with Subject[Clickable] {
  abstract override def click(): Unit = {        // ❶
    super.click()
    notifyObservers(this)
  }
}
```

❶ 请留意 abstract override 关键字，我们之前在 9.3 节曾经讨论过该关键字。

下面列出了测试脚本：

```
// src/main/scala/progscala2/objectsystem/ui/vetoable-clicks.sc
import progscala2.objectsystem.ui.Button
import progscala2.traits.ui2.{Clickable, ObservableClicks, VetoableClicks}
import progscala2.traits.observer._

val observableButton =                                            // ❶
  new Button("Okay") with ObservableClicks with VetoableClicks {
    override val maxAllowed: Int = 2                              // ❷
  }

assert(observableButton.maxAllowed == 2,                          // ❸
  s"maxAllowed = ${observableButton.maxAllowed}")

class ClickCountObserver extends Observer[Clickable] {            // ❹
  var count = 0
  def receiveUpdate(state: Clickable): Unit = count += 1
}

val clickCountObserver = new ClickCountObserver                   // ❺
observableButton.addObserver(clickCountObserver)

val n = 5
for (i <- 1 to n) observableButton.click()                        // ❻

assert(clickCountObserver.count == 2,                             // ❼
  s"count = ${clickCountObserver.count}. Should be != $n")
```

❶ 通过混入所需 trait，我们构造了一个可观察的（observable）按钮。

❷ 覆写 val 变量是我们此次练习的主要目的。请留意，此处使用 override 关键字并对 maxAllowed 变量进行完整声明是必需的。

❸ 使用 assert 语句，验证 maxAllowed 值已经成功修改了。

❹ 定义一个观察者，以追踪当前点击数。

❺ 初始化观察者实例，并将该实例“注册”到按钮主体中。

❻ 点击按钮五次。

❼ 验证观察者是否只接收到两次点击事件；其余三次点击事件都被否决了。

我们再回顾一下，trait 的混入次序决定了它们之间的优先级，我们稍后将在 11.7 一节中，完成对执行顺序的深入讲解。

如果我们尝试切换标签❶下一行中 ObservableClicks 和 VetoableClicks 的输入次序，会发生什么呢？你将看到该程序无法通过最终的断言测试，点击数将会是 5 而不是 2。为什么呢？这是因为 ObservableClicks 特征会早于 VetoableClicks 特征发现每次点击事件。换言之，VetoableClicks 现在并未做任何有意义的事情。

现在，我们知道覆写不可变字段定义。假如你希望能更灵活地控制 maxAllowed 字段，使其能够在程序运行时发生变化，那么该怎么办呢？你可以使用 var 关键字将该字段声明为可变变量，之后对 observableButton 的声明进行修改，如下所示：

```
val observableButton =
  new Button("Okay") with ObservableClicks with VetoableClicks {
    maxAllowed = 2
  }
```

此时我们已不再需要此前 observableButton 签名中出现的 override 关键字了。

考虑到逻辑一致性，你应该知道对 maxAllowed 变量进行修改后对观察者的状态会有什么影响。假如 maxAllowed 数量减少，而观察者已经统计出了一个较大的点击数，那么你是否应该引入机制减少观察者统计的数量呢？

我们现在可以讨论之前提及初始化时可能会出现的问题。在执行类体之间，该类所使用的 trait 的代码体便已经执行完毕。这也意味着特征体对字段进行初始化赋值之后，类才对该字段进行重新赋值。回顾我们之前的一个错误示例。inverse 值尚未设置便被调用。为了能够保存 maxAllowed 字段的更新信息，我们假设特征 VetoableObserver 初始化了某些私有数组。而对 maxAllowed 的最终赋值操作也许会使对象处于不一致的状态！你需要通过一些手动的方式避免这一问题，例如：直到需要对 maxAllowed 进行第一次更新时才分配存储，并确保此时已经完成了初始化过程。将 maxAllowed 字段声明为 val 字段并不能解决这一问题，尽管这一举动能提醒用户 VetoableClicks 特征已经对实例的状态进行了这样的假定，即 maxAllowed 字段状态不会被修改。除此之外，假如你是 VetoableClicks 特征的维护者，那么你需要记住一点：无论 maxAllowed 是否被声明为不可变量，用户都有可能会对该值进行覆写！

> 尽可能避免（在类中和 trait 中）使用 var 字段，而使用公共 var 字段则尤为危险。
>
> 不过，val 所提供的可见性保护并非无懈可击。我们可以在初始化子类实例时对 trait 中的 val 字段进行覆写，不过初始化完之后，该字段仍为不可变值。

### 覆写类字段

类中声明的成员，其表现与 trait 中的成员大致相同。为了能够完整地描述如何覆写类字段，下面这个示例中的继承类既覆写了 val 字段，又对 var 字段进行了重新赋值：

```
// src/main/scala/progscala2/objectsystem/overrides/class-field.sc

class C1 {
  val name = "C1"
  var count = 0
}

class ClassWithC1 extends C1 {
  override var name = "ClassWithC1"
  count = 1
}

val c = new ClassWithC1()
println(c.name)
println(c.count)
```

覆写具体 val 字段时，override 关键字是必不可少的，不过对于 count 这个 var 字段而言，则不然。这是因为修改 val 字段意味着我们正在修改某一常量（val 字段）的初始化过程，这类“特殊”操作需要使用 override 关键字。

运行该脚本后会产生下列输出：

```
ClassWithC1
1
```

正如我们所预计的那样，继承类对这两个字段都进行了覆写。下面对之前的示例进行修改，将基类中的 val 字段和 var 字段都修改成 abstract 字段：

```
// src/main/scala/progscala2/objectsystem/overrides/class-abs-field.sc

abstract class AbstractC1 {
  val name: String
  var count: Int
}

class ClassWithAbstractC1 extends AbstractC1 {
  val name = "ClassWithAbstractC1"
  var count = 1
}

val c = new ClassWithAbstractC1()
println(c.name)
println(c.count)
```

由于这些字段均声明为 abstract 类型，因此 ClassWithAbstractC1 中不再需要 override 关键字。脚本输出如下所示：

```
ClassWithAbstractC1
1
```

有必要强调一下：name 和 count 是抽象字段，它们并不是包含默认值的具体字段。如果在 Java 类中进行类似的声明，如 String name，Java 将会声明一个具有默认值的具体字段，而在本例中默认值为 null。Java 并不支持抽象字段，只支持抽象方法。

## 11.5 覆写抽象类型

我们在 2.13 节中介绍了什么是抽象类型，而 Java 并不支持抽象类型。我们回顾一下相关章节中出现过的 BulkReader 示例：

```
abstract class BulkReader {
  type In
  val source: In
  def read: String  // 该方法会读取数据源内容,并返回字符串
}

class StringBulkReader(val source: String) extends BulkReader {
  type In = String
  def read: String = source
}
...
```

该示例演示了如何声明抽象类型以及如何在继承类中为该抽象类型定义具体值。BulkReader 类声明了 In 类型，但却未初始化该类型。而具体继承类 StringBulkReader 使用 type In = String 语句为该类型提供了具体值。

不同于字段和方法，我们无法对具体的类型定义进行覆写。

## 11.6 无须区分访问方法和字段：统一访问原则

我们将再次聚焦 VetoableClick 特征中的 maxAllowed 字段，并将讨论一种有趣的设计方式：将继承和统一访问原则混合在一起。我们之前在 8.6.1 一节中已经对该原则进行了讲解。

下面列举了 VetoableClick 的一类新的实现，我们称之为 VetoableClickUAP（UAP 是 uniform access principle 的缩写，即统一访问原则）：

```
// src/main/scala/progscala2/objectsystem/ui/vetoable-clicks-uap.sc
import progscala2.objectsystem.ui.Button
import progscala2.traits.ui2.{Clickable, ObservableClicks, VetoableClicks}
import progscala2.traits.observer._

trait VetoableClicksUAP extends Clickable {

  def maxAllowed: Int = 1                         // ❶

  private var count = 0

  abstract override def click() = {
    if (count < maxAllowed) {
      count += 1
      super.click()
```

```
    }
  }
}

val observableButton =
  new Button("Okay") with ObservableClicks with VetoableClicksUAP {
    override val maxAllowed: Int = 2            // ❷
  }

assert(observableButton.maxAllowed == 2,
  s"maxAllowed = ${observableButton.maxAllowed}")

class ClickCountObserver extends Observer[Clickable] {
  var count = 0
  def receiveUpdate(state: Clickable): Unit = count += 1
}

val clickCountObserver = new ClickCountObserver
observableButton.addObserver(clickCountObserver)

val n = 5
for (i <- 1 to n) observableButton.click()

assert(clickCountObserver.count == 2,
  s"count = ${clickCountObserver.count}. Should be != $n")
```

❶ 我们将 `maxAllowed` 定义成一个默认返回值为 `1` 的方法。

❷ 我们并没有对 `maxAllowed` 方法进行覆写，而是使用了一个值定义（`val` 类型）对其重新定义。

由于我们已经定义了 `maxAllowed` 方法，因此必须使用 `override` 关键字。假如特征体中的 `maxAllowed` 方法为抽象方法，那么 `override` 关键字便不是必须的。

该脚本的输出与之前相同，不过我们利用统一访问原则把方法定义覆写为值定义。那么为什么 Scala 会允许这种行为呢？

声明某一函数时，只要函数实现能正确执行，该函数就有权在每次调用时返回不同的结果值。不过，假如该函数每次只返回一个特定值的话，该函数声明便具备了一致性。当然，函数式编程比较青睐这种行为。而理想状态下，在纯函数编程的世界里，无参方法总是应该返回相同值。

所以，假如将函数调用替换成某一个值时，我们利用了*引用透明性*（referential transparency）原则，并未违背任何方法实现应遵循的规则。

基于这一原因，Scala 库中定义的 trait 中常常使用无参方法，而不是字段值。如果愿意的话，你也可以将这些方法视为*属性读取器*（property reader）。这样一来，该类型的开发人员便能非常灵活地实现该方法。开发人员可以将代价昂贵的初始化过程推迟到必须处理的时候再处理，也可以简单地为该方法设置一个返回值。

与 Java 代码进行互操作时，你可以在子类中将方法覆写为值，这样能为开发人员带来便利。例如：你可以对某些 getter 方法进行覆写，将其修改为 `val` 值并放到构造函数中即可

（Scala 会为构造函数参数表中出现的参数自动生成 getter 方法）。

在下面这个示例中，Scala 语言中的 `Person` 类从某些遗留的 Java 库中继承了 `PersonInterface` 类：

```
class Person(val getName: String) extends PersonInterface
```

如果你只需要对 Java 代码中的某些访问方法进行覆写，那么使用这项技术能帮助你很快地完成工作。

我们能不能使用 `var` 变量对无参方法进行覆写呢？能否使用方法对某一 `val` 值或 `var` 值进行覆写呢？由于在这两类覆写中，覆写后的类型行为无法匹配之前的行为，因此这是不被允许的。

如果你尝试使用 `var` 变量覆写无参方法，系统将返回错误：写方法 `override name_=` 无法覆写任何方法。举个例子，假如我们在某个 trait 中使用 `def name: String` 语句声明抽象方法，之后在实现子类中使用 `val name="foo"` 语句对该方法进行覆写。这等同于覆写了两个方法，trait 中原先定义的方法和 `def name_=(...)` 方法，但是 trait 中并不存在 `name_` 方法。

如果 Scala 允许使用方法对 `val` 值进行覆写，为了与 `val` 的语法保持一致性，Scala 需要保证每次调用该方法总能返回相同值，但 Scala 无法做到这点。

## 11.7　对象层次结构的线性化算法

由于采用了单继承模型，假如我们不使用可混入的 trait，继承层次结构将会变成一条直线，将各个祖先节点依次连接。如果引入 trait 的话，由于每个类型都有可能继承自其他 trait 或类，继承层次关系便形成了一个有向无环图。

“线性化算法”是一类用于对层级结构图进行“扁平化”处理的算法，引入该算法是为了解决方法查找的优先级问题、构造函数调用顺序、`super` 关键字绑定问题等一系列问题。

我们之前在 9.3 节中曾看到过混入了多个 trait 的实例，其中 Scala 将按照从右到左的声明顺序对这些 trait 进行绑定。而下列示例也证实了这一条简单的线性化规则：

```
// src/main/scala/progscala2/objectsystem/linearization/linearization1.sc

class C1 {
  def m = print("C1 ")
}

trait T1 extends C1 {
  override def m = { print("T1 "); super.m }
}

trait T2 extends C1 {
  override def m = { print("T2 "); super.m }
}

trait T3 extends C1 {
  override def m = { print("T3 "); super.m }
```

```
}

class C2 extends T1 with T2 with T3 {
  override def m = { print("C2 "); super.m }
}

val c2 = new C2
c2.m
```

该脚本执行后将输出以下信息：

```
C2 T3 T2 T1 C1
```

由此，我们可以看出 trait 中的 m 方法将依照声明顺序，从右到左的被调用。我们稍后也将解释为什么 C1 会出现在列表的末尾处。

接下来，我们将会看到构造函数的调用顺序：

```
// src/main/scala/progscala2/objectsystem/linearization/linearization2.sc

class C1 {
  print("C1 ")
}

trait T1 extends C1 {
  print("T1 ")
}

trait T2 extends C1 {
  print("T2 ")
}

trait T3 extends C1 {
  print("T3 ")
}

class C2 extends T1 with T2 with T3 {
  println("C2 ")
}

val c2 = new C2
```

执行该脚本后将产生以下输出信息：

```
C1 T1 T2 T3 C2
```

我们可以发现，构造顺序与之前的方法调用顺序恰恰相反。由于继承类型在构造过程中常常需要使用父类型的字段和方法，而这种构造函数调用顺序恰恰能够确保父类型会先于继承类型被构造。

第一段线性化脚本的输出结果的末尾处实际上缺少了两个类型。对引用类型执行线性化算法后，结果的末尾处应包含 AnyRef 类型和 Any 类型。因此，对 C2 做线性化处理后的实际输出如下所示：

```
C2 T3 T2 T1 C1 AnyRef Any
```

在 Scala 2.10 出现之前，Scala 中还存在一个用于标记的特征 ScalaObject，该 trait 在类层次关系表中位于 AnyRef 之前。因此 AnyRef 和 Any 类型中并未使用我们所使用的 print 语句，所以我们的输出结果中当然不会显示它们。

与引用类型不同，值类型是 AnyVal 类的子类，它们均被声明为 abstract final 类型。编译器负责初始化这些值类型。由于我们无法编写这些值类型的子类，因此这些值类型的线性化算法非常简单直白。

那些新创建的价值类（value class），它们的线性化算法是什么样的呢？我们将对之前出现的 USPhoneNumber 进行修改，在类中添加 m 方法，该方法与我们之前使用的 m 方法完全一样。价值类不允许我们在类型体中添加之前的 print 语句：

```
// src/main/scala/progscala2/basicoop/valueclassphoneNumber.sc

class USPhoneNumber(val s: String) extends AnyVal {

  override def toString = {
    val digs = digits(s)
    val areaCode  = digs.substring(0,3)
    val exchange  = digs.substring(3,6)
    val subnumber = digs.substring(6,10)  // "subscriber number"
    s"($areaCode) $exchange-$subnumber"
  }

  private def digits(str: String): String = str.replaceAll("""\D""", "")
}

val number = new USPhoneNumber("987-654-3210")
// Result: number: USPhoneNumber = (987) 654-3210
```

调用 m 方法后，该示例将打印下列信息：

```
USPhoneNumber Formatter Digitizer M
```

我们之前曾看到过一组类层次体系，该体系中每个类都被命名为 C*。而上述示例的输出结果与 C* 类层次体系打印的结果一致。不过请注意，上述示例将 C* 体系结构中出现的 M 特征混入到一些其他的 trait 中。为什么 M 出现在输出信息的末尾位置呢，是不是表明定义在 M 特征中的 m 方法会被最后找到呢？我们将对线性化算法进行更深入地学习。

我们将使用一组 C* 类对线性化算法进行讲解。在这组类中，所有的类和 trait 均定义了方法 m。由于下面示例中的实例的类型是 C2 类型，因此 C2 类中定义的 m 方法会被最早调用。C2.m 方法调用了 super.m 方法，该方法将被解析为 T3.m 方法。对输出结果进行分析，发现方法查找似乎并未采用深度优先算法，而是采用了广度优先算法。假如查找方法时采取了深度优先算法，那么会先执行 T3.m，再执行 C1.m，接着是 T3.m、T2.m 以及 T1.m，最后则是 C1.m。C1 是这三个 trait 的父类，那么我们会通过哪个 trait 引导到 C1 呢？事实上，正如我们将看到的那样，Scala 将使用广度优先的方式查找方法，并“延后”执行已查找到的方法。下面，我们将对第一个示例进行修改，并更仔细的观察我们是如何找到 C1 类的：

```
// src/main/scala/progscala2/objectsystem/linearization/linearization3.sc
```

```
class C1 {
  def m(previous: String) = print(s"C1($previous)")
}

trait T1 extends C1 {
  override def m(p: String) = { super.m(s"T1($p)") }
}

trait T2 extends C1 {
  override def m(p: String) = { super.m(s"T2($p)") }
}

trait T3 extends C1 {
  override def m(p: String) = { super.m(s"T3($p)") }
}

class C2 extends T1 with T2 with T3 {
  override def m(p: String) = { super.m(s"C2($p)") }
}

val c2 = new C2
c2.m("")
```

现在我们将 super.m 的调用者名称作为参数传入各个 m 方法中，之后 C1 便会打印出调用该方法的对象名。执行上述脚本将生成下列输出：

```
C1(T1(T2(T3(C2()))))
```

下面列出了类型线性化的具体算法。如果想了解更官方的定义，请查找“Scala 语言规范”（http://www.scala-lang.org/docu/files/ScalaReference.pdf）相关内容。

**线性化算法**

(1) 当前实例的具体类型会被放到线性化后的首个元素位置处。

(2) 按照该实例父类型的顺序从右到左的放置节点，针对每个父类型执行线性化算法，并将执行结果合并。（我们暂且不对 AnyRef 和 Any 类型进行处理。）

(3) 按照从左到右的顺序，对类型节点进行检查，如果类型节点在该节点右边出现过，那么便将该类型移除。

(4) 在类型线性化层次结构末尾处添加 AnyRef 和 Any 类型。

如果是对价值类执行线性化算法，请使用 AnyVal 类型替代 AnyRef 类型。

线性化算法能说明之前示例中是如何从 T1 查找到 C1 的。C1 同样出现在 T3 和 T2 的线性化层次结构中，不过由于 T3 和 T2 出现的时间早于 T1，因此这两个类能为 T1 提供的查找类型便从线性化列表中删除了。与之相似，因为相同的原因，USPhoneNumber 示例中出现的 M 特征最后出现在了线性化列表中的右侧。

下面，我们将使用一个略微复杂一些的示例对线性化算法进行讲解：

```scala
// src/main/scala/progscala2/objectsystem/linearization/linearization4.sc

class C1 {
  def m = print("C1 ")
}

trait T1 extends C1 {
  override def m = { print("T1 "); super.m }
}

trait T2 extends C1 {
  override def m = { print("T2 "); super.m }
}

trait T3 extends C1 {
  override def m = { print("T3 "); super.m }
}

class C2A extends T2 {
  override def m = { print("C2A " ); super.m }
}

class C2 extends C2A with T1 with T2 with T3 {
  override def m = { print("C2 "); super.m }
}

def calcLinearization(obj: C1, name: String) = {
  print(s"$name: ")
  obj.m
  print("AnyRef ")
  println("Any")
}

calcLinearization(new C2, "C2 ")
println("")
calcLinearization(new T3 {}, "T3 ")
calcLinearization(new T2 {}, "T2 ")
calcLinearization(new T1 {}, "T1 ")
calcLinearization(new C2A, "C2A")
calcLinearization(new C1, "C1 ")
```

输出信息如下：

```
C2 : C2 T3 T1 C2A T2 C1 AnyRef Any

T3 : T3 C1 AnyRef Any
T2 : T2 C1 AnyRef Any
T1 : T1 C1 AnyRef Any
C2A: C2A T2 C1 AnyRef Any
C1 : C1 AnyRef Any
```

为了帮助大家理解，我们计算出了其他类型的线性化层次结构；与此同时，为了提醒自己 AnyRef 和 Any 类型也应出现在线性化列表中，我们也添加了这两个类的信息。

下面让我们对 C2 应用线性化算法，并对结果进行确认。为了使计算过程更加清楚，我们

忽略了 AnyRef 和 Any 类，在计算的最后才把这两个类添加上。具体过程，请查看表 11-1。

表 11-1：C2类型的线性化计算过程，C2类型定义为：C2 extends C2A with T1 with T2 with T3 {…}

| # | 线性化层次结构 | 描　述 |
|---|---|---|
| 1 | C2 | 添加当前实例类型 |
| 2 | C2，T3，C1 | 添加 T3 的线性化列表（离 T3 最远的元素放在列表的右侧） |
| 3 | C2，T3，C1，T2，C1 | 添加 T2 的线性化列表 |
| 4 | C2，T3，C1，T2，C1，T1，C1 | 添加 T1 的线性化列表 |
| 5 | C2，T3，C1，T2，C1，T1，C1，C2A，T2，C1 | 添加 C2A 的线性化列表 |
| 6 | C2，T3，T2，T1，C2A，T2，C1 | 移除所有重复的 C1 元素，只保留最后一个 |
| 7 | C2，T3，T1，C2A，T2，C1 | 移除所有重复的 T2 元素，只保留最后一个 |
| 8 | C2，T3，T1，C2A，T2，C1，AnyRef，Any | 完毕！ |

算法做的事情其实就是：首先将子类都放置好，再将共享类型放到线性化列表的右端。

请对脚本进行修改，尝试使用一个不同的继承结构，看看你能否使用线性化算法重新推导出结果。

过于复杂的类型继承结构会导致方法查找产生令人“意外”的结果。如果你希望通过线性化算法了解方法查找的顺序，请先简化代码。

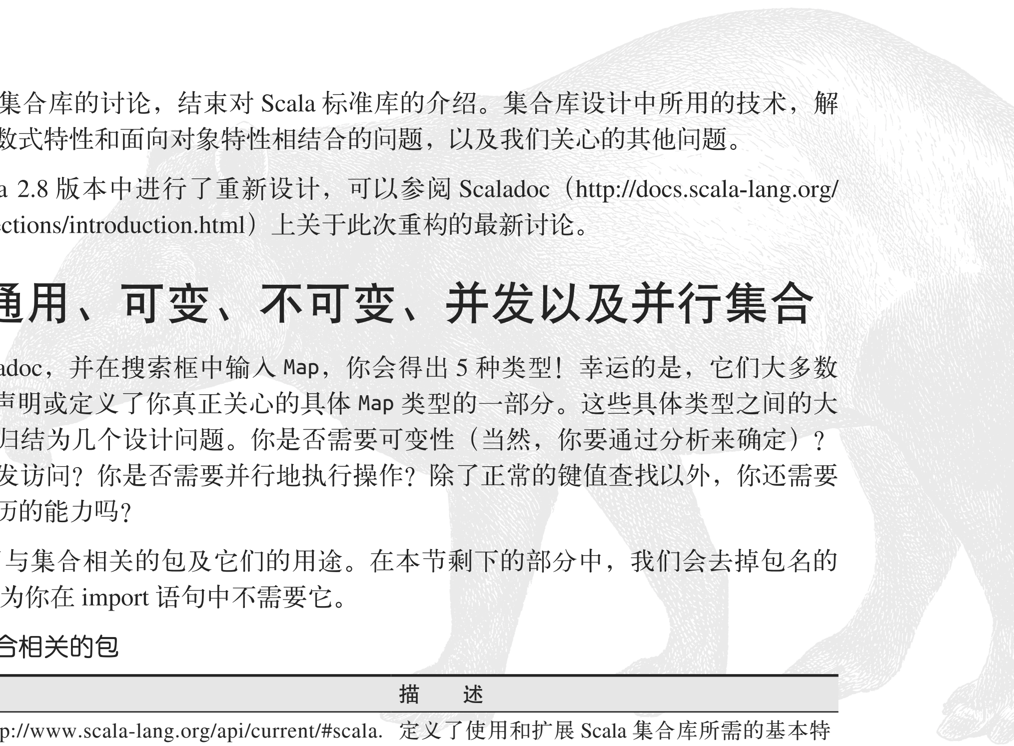
## 11.8　本章回顾与下一章提要

我们已经学习了在继承类中对父类成员进行覆写所带来的好处，其中提及了 Scala 允许我们使用价值类型对无参方法进行覆写。最后，我们介绍了 Scala 用于解决成员查找问题的线性化算法的一些细节。

在下一章中，我们将学习 Scala 的集合库。

# 第 12 章

# Scala集合库



我们将通过对集合库的讨论，结束对 Scala 标准库的介绍。集合库设计中所用的技术，解决了如何将函数式特性和面向对象特性相结合的问题，以及我们关心的其他问题。

集合库在 Scala 2.8 版本中进行了重新设计，可以参阅 Scaladoc（http://docs.scala-lang.org/overviews/collections/introduction.html）上关于此次重构的最新讨论。

## 12.1 通用、可变、不可变、并发以及并行集合

如果打开 Scaladoc，并在搜索框中输入 `Map`，你会得出 5 种类型！幸运的是，它们大多数是 trait，且只声明或定义了你真正关心的具体 `Map` 类型的一部分。这些具体类型之间的大部分差异可以归结为几个设计问题。你是否需要可变性（当然，你要通过分析来确定）？你是否需要并发访问？你是否需要并行地执行操作？除了正常的键值查找以外，你还需要按固定顺序遍历的能力吗？

表 12-1 列出了与集合相关的包及它们的用途。在本节剩下的部分中，我们会去掉包名的 scala 前缀，因为你在 import 语句中不需要它。

表12-1：与集合相关的包

| 名　称 | 描　述 |
|---|---|
| `collection`（http://www.scala-lang.org/api/current/#scala.collection.package） | 定义了使用和扩展 Scala 集合库所需的基本特征和对象，包括子包中的所有定义。你需要用到的大部分抽象都在这里定义 |
| `collection.concurrent`（http://www.scala-lang.org/api/current/#scala.collection.concurrent.package） | 定义了一个 `Map` 特征和具有原子、无锁操作特性的 `TrieMap` 类 |

（续）

| 名　　称 | 描　　述 |
| --- | --- |
| collection.convert（http://www.scala-lang.org/api/current/#scala.collection.convert.package） | 定义了用 Java 集合抽象来包装 Scala 集合的类型，以及用 Scala 集合抽象包装 Java 集合的类型 |
| collection.generic（http://www.scala-lang.org/api/current/#scala.collection.generic.package） | 定义了用来构建特定集合（如可变集合、不可变集合等）的可重用组件 |
| collection.immutable（http://www.scala-lang.org/api/current/#scala.collection.immutable.package） | 定义了你最常用的不可变集合 |
| collection.mutable（http://www.scala-lang.org/api/current/#scala.collection.mutable.package） | 定义了可变集合，大部分特定集合类型都以可变或不可变的形式存在，但并非全部 |
| collection.parallel（http://www.scala-lang.org/api/current/#scala.collection.parallel.package） | 定义了用来构建特定集合（如可变集合、不可变集合等）的可重用组件，可将处理任务分发给并行的多个线程 |
| collection.parallel.immutable（http://www.scala-lang.org/api/current/#scala.collection.parallel.immutable.package） | 定义了支持并行的不可变集合 |
| collection.parallel.mutable（http://www.scala-lang.org/api/current/#scala.collection.parallel.mutable.package） | 定义了支持并行的可变集合 |
| collection.script（http://www.scala-lang.org/api/current/#scala.collection.script.package） | 已经废弃的包，包含了用来观察集合操作的工具 |

我们不会讨论以上这些包中定义的大部分类型，但我们要讨论每个包中最重要的方面。被废弃的 collection.script 不在继续讨论之列。

## 12.1.1　scala.collection包

在 collection 包中声明的类型定义了可变及不可变的序列、可变及不可变的并行、并发集合类型共享的抽象，其中有的不仅声明，而是直接进行了定义。这就意味着，比如只能在可变类型中使用的带破坏性的（可变的）操作不是在这里定义的。不过，在运行时如果集合是可变的，我们可能要考虑线程的安全问题。

回顾一下 6.7.1 节，从 Predef 得到的 Seq 类型是 collection.Seq（http://www.scala-lang.org/api/current/#scala.collection.Seq），而 Predef 引入的其他公共类型是以 collection.immutable 开头的，如 List、Map 和 Set。Predef 使用 collection.Seq 的原因是要让 Scala 可以像处理序列一样处理 Java 的数组，而 Java 的数组是可变的（Predef 事实上定义了从 Java 数组到 collection.mutable.ArrayOps（http://www.scala-lang.org/api/current/index.html#scala.collection.mutable.ArrayOps）的隐式转换，而后者支持序列的相关操作）。Scala 计划在未来的版本中用不可变的 Seq 代替它。

不幸的是，就目前而言，这也意味着如果一个方法声明它返回一个序列，它可能会返回一个可变的序列实例。同样地，如果一个方法需要一个序列参数，调用者也可以传入一个可变的序列实例。

如果你更喜欢用更安全的 immutable.Seq 作为默认的 Seq，常见的方法是，为你的项目定义

一个包对象，其中定义了 Seq 类型，以覆盖 Predef 定义的 Seq，如下所示：

```
// src/main/scala/progscala2/collections/safeseq/package.scala
package progscala2.collections
package object safeseq {
  type Seq[T] = collection.immutable.Seq[T]
}
```

然后，只在需要的时候导入以上内容即可。注意观察以下 REPL 会话中，Seq 行为的变化：

```
// src/main/scala/progscala2/collections/safeseq/safeseq.sc

scala> val mutableSeq1: Seq[Int] = List(1,2,3,4)
mutableSeq1: Seq[Int] = List(1, 2, 3, 4)

scala> val mutableSeq2: Seq[Int] = Array(1,2,3,4)
mutableSeq2: Seq[Int] = WrappedArray(1, 2, 3, 4)

scala> import progscala2.collections.safeseq._
import progscala2.collections.safeseq._

scala> val immutableSeq1: Seq[Int] = List(1,2,3,4)
immutableSeq1: safeseq.Seq[Int] = List(1, 2, 3, 4)

scala> val immutableSeq2: Seq[Int] = Array(1,2,3,4)
<console>:10: error: type mismatch;
 found   : Array[Int]
 required: safeseq.Seq[Int]
    (which expands to)  scala.collection.immutable.Seq[Int]
       val immutableSeq2: Seq[Int] = Array(1,2,3,4)
                                          ^
```

前两个 Seq 是由 Predef 暴露的默认项 collection.Seq。第一个 Seq 引用了一个不可变列表，第二个 Seq 引用了可变的（经过包装的）Java 数组。

然后我们导入了新的 Seq 定义，从而遮蔽了 Predef 中的 Seq 定义。

现在 Seq 实际上是 safeseq.Seq 的别名，我们不能用它来引用数组，因为别名 immutable.Seq 不能引用一个可变的集合。

无论哪种方式，如果我们想取集合的前几个元素或希望从集合的一端遍历到另一端，Seq 都是具体集合的一个方便、好用的抽象。

### 12.1.2　collection.concurrent包

这个包只定义了两种类型：collection.concurrent.Map（http://www.scala-lang.org/api/current/#scala.collection.concurrent.Map）特征和实现了该 trait 的 collection.concurrent.TrieMap（http://www.scala-lang.org/api/current/#scala.collection.concurrent.TrieMap）类。

Map 继承了 collection.mutable.Map（http://www.scala-lang.org/api/current/#scala.collection.mutable.Map），但它使用了原子操作，因此得以支持线程安全的并发访问。

collection.mutable.Map 的实现是一个字典树散列类 collection.concurrent.TrieMap。它

实现了并发、无锁的散列数组，其目的是支持可伸缩的并发插入和删除操作，并提高内存使用效率。

## 12.1.3 collection.convert包

在这个包中定义的类型是用来实现隐式转换方法的，隐式转换将 Scala 的集合包装为 Java 集合，反之亦然。我们在 5.7 节对此有过讨论。

## 12.1.4 collection.generic包

collection 包声明的抽象适用于所有集合，而 collection.generic 只为实现特定的可变、不可变、并行及并发集合提供一些组件。这里的大多数类型只对集合的实现者有意义。

## 12.1.5 collection.immutable包

大部分时间你都会与 immutable 包中定义的集合打交道。这些类型提供了单线程（与并行相对）操作，由于类型是不可变的，因而是线程安全的。表 12-2 按字母顺序列出了这个包中最常用的集合类型。

表12-2：最常用的不可变集合

| 名　称 | 描　述 |
|---|---|
| BitSet（http://www.scala-lang.org/api/current/#scala.collection.immutable.BitSet） | 非负整数的集合，内存效率高。元素表示为可变大小的比特数组，其中比特被打包为 64 比特的字。最大元素个数决定了内存占用量 |
| HashMap（http://www.scala-lang.org/api/current/#scala.collection.immutable.HashMap） | 用字典散列实现的映射表 |
| HashSet（http://www.scala-lang.org/api/current/#scala.collection.immutable.HashSet） | 用字典散列实现的集合 |
| List（http://www.scala-lang.org/api/current/#scala.collection.immutable.List） | 用于相连列表的 trait，头节点访问复杂度为 $O(1)$，其他元素为 $O(n)$。其伴随对象有 apply 方法和其他“工厂”方法，可以用来构造 List 的子类实例 |
| ListMap（http://www.scala-lang.org/api/current/#scala.collection.immutable.ListMap） | 用列表实现的不可变映射表 |
| ListSet（http://www.scala-lang.org/api/current/#scala.collection.immutable.ListSet） | 用列表实现的不可变集合 |
| Map（http://www.scala-lang.org/api/current/#scala.collection.immutable.Map） | 为所有不可变的映射表定义的 trait，随机访问复杂度为 $O(1)$，其伴随对象有 apply 方法和其他“工厂”方法，可以用来构造其子类实例 |
| Nil（http://www.scala-lang.org/api/current/#scala.collection.immutable.Nil$） | 用来表示空列表的对象 |
| NumericRange（http://www.scala-lang.org/api/current/#scala.collection.immutable.NumericRange） | Range 类的推广版本，将适用范围推广到任意完整的类型。使用时，必须提供类型的完整实现 |
| Queue（http://www.scala-lang.org/api/current/#scala.collection.immutable.Queue） | 不可变的 FIFO（先入先出）队列 |

（续）

| 名　　称 | 描　　述 |
| --- | --- |
| Seq（http://www.scala-lang.org/api/current/#scala.collection.immutable.Seq） | 为不可变序列定义的 trait，其伴随对象有 apply 方法和其他“工厂”方法，可以用来构造其子类实例 |
| Set（http://www.scala-lang.org/api/current/#scala.collection.immutable.Set） | 特征，为不可变集合定义了操作，其伴随对象有 apply 方法和其他“工厂”方法，可以用来构造其子类实例 |
| SortedMap（http://www.scala-lang.org/api/current/#scala.collection.immutable.Set） | 为不可变映射表定义的 trait，包含一个可按特定排列顺序遍历元素的迭代器。其伴随对象有 apply 方法和其他“工厂”方法，可以用来构造其子类实例 |
| SortedSet（http://www.scala-lang.org/api/current/#scala.collection.immutable.SortedSet） | 为不可变集合定义的 trait，包含一个可按特定排列顺序遍历元素的迭代器。其伴随对象有 apply 方法和其他“工厂”方法，可以用来构造其子类实例 |
| Stack（http://www.scala-lang.org/api/current/#scala.collection.immutable.Stack） | 不可变的 LIFO（后入先出）栈 |
| Stream（http://www.scala-lang.org/api/current/#scala.collection.immutable.Stream） | 对元素惰性求值的列表，可以支持拥有无限个潜在元素的序列 |
| TreeMap（http://www.scala-lang.org/api/current/#scala.collection.immutable.TreeMap） | 不可变映射表，底层用红黑树实现，操作的复杂度为 $O(\log(n))$ |
| TreeSet（http://www.scala-lang.org/api/current/#scala.collection.immutable.TreeSet） | 不可变集合，底层用红黑树实现，操作的复杂度为 $O(\log(n))$ |
| Vector（http://www.scala-lang.org/api/current/#scala.collection.immutable.Vector） | 不可变、支持下标的序列的默认实现 |

Bitset 是非负整数的集合，被打包为 64 比特的字组成的大小可变的数组。最大元素个数决定了内存占用量。

Vector 使用基于树的、持久的数据结构来实现，如 6.11 节讨论的那样，它可以提供出色的性能，操作的均摊复杂度达到 $O(1)$。

Map 的源代码（https://github.com/scala/scala/blob/v2.11.2/src/library/scala/collection/immutable/Map.scala#L1）值得一读，尤其是伴随对象的部分。需要注意，Map 还为只有 0 到 4 个键值的情况声明了专用的实现。当你调用 Map.apply（定义于父特征）时，Scala 会创建一个最适合当前数据的 Map 实例。

### 12.1.6　scala.collection.mutable包

有些时候你需要一个在单线程操作中的可变集合类型。我们已经讨论了不可变集合为何应该成为默认选项的问题。对这些集合做可变操作不是线程安全的。然而，为了提高性能等原因，有原则、谨慎地使用可变数据也是恰当的。表 12-3 按字母顺序给出了 mutable 包中最常用的集合。

表12-3：最常用的可变集合

| 名　　称 | 描　　述 |
| --- | --- |
| AnyRefMap | 为 AnyRef 类型的键准备的映射表，采用开放地址法解决冲突。大部分操作通常都比 HashMap 快 |
| ArrayBuffer | 内部用数组实现的缓冲区类，追加、更新与随机访问的均摊时间复杂度为 $O(1)$，头部插入和删除操作的复杂度为 $O(n)$ |
| ArrayOps | Java 数组的包装类，实现了序列操作 |
| ArrayStack | 数组实现的栈，比通用的栈速度快 |
| BitSet | 内存效率高的非负整数集合，见表 12-2 对 immutable.BitSet 的讨论 |
| HashMap | 基于散列表的可变版本的映射 |
| HashSet | 基于散列表的可变版本的集合 |
| HashTable | 用于实现基于散列表的可变集合的 trait |
| ListMap | 基于列表实现的映射 |
| LinkedHashMap | 基于散列表实现的映射，元素可以按其插入顺序进行遍历 |
| LinkedHashSet | 基于散列表实现的集合，元素可以按其插入顺序进行遍历 |
| LongMap | 键的类型为 Long，基于散列表实现的可变映射，采用开放地址法解决冲突。大部分操作都比 HashMap 快 |
| Map | Map 特征的可变版，其伴随对象有 apply 方法和其他“工厂”方法，可以用来构造 List 的子类实例 |
| MultiMap | 可变的映射，可以对同一个键赋以多个值 |
| PriorityQueue | 基于堆的，可变优先队列。对于类型为 A 的元素，必须存在隐含的 Ordering[A] 实例。 |
| Queue | 可变的 FIFO（先入先出）队列 |
| Seq | 表示可变序列的 trait，其伴随对象有 apply 方法和其他“工厂”方法，可以用来构造 List 的子类实例 |
| Set | 声明了可变集合相关操作的 trait，其伴随对象有 apply 方法和其他“工厂”方法，可以用来构造 List 的子类实例 |
| SortedSet | 表示可变集合的 trait，包含一个可按特定排列顺序遍历元素的迭代器。其伴随对象有 apply 方法和其他“工厂”方法，可以用来构造 List 的子类实例 |
| Stack | 可变的 LIFO（后入先出）栈 |
| TreeSet | 可变集合，底层用红黑树实现，操作的复杂度为 $O(\log(n))$ |
| WeakHashMap | 可变的散列映射，引用元素时采用弱引用。当元素不再有强引用时，就会被删除。该类包装了 WeakHashMap |
| WrappedArray | Java 数组的包装类，支持序列的操作 |

WrappedArray 与 ArrayOps 差不多完全相同，差别仅在于它们各自返回 Array 的方法上。对于 ArrayOps，返回的是新的 Array[T]，而 WrappedArray 返回的是新的 WrappedArray[T]。所以，如果用户需要 Array，更适合用 ArrayOps；但当用户并不关心这一点的时候，如果涉及序列转换，使用 WrappedArray 会更加高效。这是因为 WrappedArray 避免了 ArrayOps 中对数组的“打包”和“分拆”工作。

### 12.1.7 scala.collection.parallel包

并行集合的思想是利用现代多核系统提供的并行硬件多线程。根据定义，任何可以并行指定的集合操作都可以利用这种并行性。

具体地说，集合被分成多个片段，操作（如 map）应用在各个片段上，然后将结果组合在一起，形成最终结果。也就是说，这里用了分而治之的策略。

在实践中，并行集合没有被广泛使用，因为在许多情况下，并行化的开销可能会掩盖它的优点，而且不是所有的操作都可以并行执行。开销包括线程调度、数据分块、以及最后对结果的合并。通常情况下，除非该集合规模极大，否则串行执行速度会更快。所以，一定要仔细评估真实世界里的场景，确定集合是否足够大，并行操作是否足够快，来让我们选择并行集合。

对于具体的并行集合类型，你可以直接用与非并行集合相同的惯例来实例化它，也可以对相应的非并行集合调用 par 方法。

并行集合的组合也与非并行集合类似。它们在 scala.collection.parallel 包中具有共同的 trait 和类，在 immutable 子包中定义了相同的不可变具体集合，在 mutable 子包中定义了相同的可变具体集合。

最后，有一点有必要理解，并行意味着嵌套操作的顺序是未定义的。考虑如下示例，我们将从 1 到 10 的数字连接起来放进一个字符串中：

```
// src/main/scala/progscala2/collections/parallel.sc

scala> ((1 to 10) fold "") ((s1, s2) => s"$s1 - $s2")
res0: Any = " - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10"

scala> ((1 to 10) fold "") ((s1, s2) => s"$s1 - $s2")
res1: Any = " - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10"

scala> ((1 to 10).par fold "") ((s1, s2) => s"$s1 - $s2")
res2: Any = " - 1 -  - 2 -  - 3 - 4 - 5 -  - 6 -  - 7 -  - 8 -  - 9 -  - 10"

scala> ((1 to 10).par fold "") ((s1, s2) => s"$s1 - $s2")
res3: Any = " - 1 -  - 2 -  - 3 -  - 4 - 5 -  - 6 -  - 7 -  - 8 - 9 - 10"
```

对于非并行的版本，代码总是能返回相同的结果。但对于并行版本，多次运行会返回不同的结果！

然而，加法能够正常工作：

```
scala> ((1 to 10) fold 0) ((s1, s2) => s1 + s2)
res4: Int = 55

scala> ((1 to 10) fold 0) ((s1, s2) => s1 + s2)
res5: Int = 55

scala> ((1 to 10).par fold 0) ((s1, s2) => s1 + s2)
res6: Int = 55
```

```
scala> ((1 to 10).par fold 0) ((s1, s2) => s1 + s2)
res7: Int = 55
```

每次运行都返回了相同的结果。

具体地讲，操作必须满足结合律，才能在并行操作中返回稳定的、可预测的结果。也就是说，(a+b)+c==a+(b+c) 必须始终成立。并行运行时，每次输出的字符串中，空格和 - 分隔符数量不一致，这表明这里使用的操作不满足结合律。每次运行，并行集合都被按照不同的、不可预测的方式拆分。

除了满足结合律外，加法还满足交换律，但这不是必须的。注意到，在字符串的例子中，各个元素按照可预见的、从左到右的顺序排列，说明交换律并不是必要的。

由于并行集合都有非并行的对应类型，后者我们已经讨论过了，所以我就不列举非并行集合的具体类型了。关于 parallel、parallel.immutable 和 parallel.mutable 包，我们可以参考 Scaladoc。此外，Scaladoc 还讨论了这里未讨论的其他问题。

## 12.2　选择集合

除了决定使用可变集合还是不可变集合，平行集合或非并行集合，在给定的场景下，你还需要决定究竟应该选择什么集合类型?

这里有一些非正式的标准和方案可供考虑。集合类型的不同操作复杂度值得研究。在 Scaladoc（http://docs.scala-lang.org/overviews/collections/performance-characteristics.html）中有一个详尽的列表。在 StackOverflow（http://stackoverflow.com/questions/24464792/scala-collections-flowchart/24465514#24465514）上也有一个关于选择集合类型的讨论。

当可能同时有可变和不可变两种选择时，我将使用 immutable.List（mutable.LinkedList）来表示不可变（可变）的选项。

你需要有序、可遍历的序列吗？那就考虑 immutable.List（mutable.LinkedList）、immutable.Vector 或 mutable.ArrayBuffer。

List 提供了 $O(1)$ 的前插和取头节点复杂度，但追加和读取内部其他元素的复杂度为 $O(n)$。

由于 Vector 是可持久的数据结构（如前面所讨论的一样），它的所有操作都是 $O(1)$ 复杂度。

如果你需要随机访问，ArrayBuffer 是更好的选择。追加、更新和随机访问所需要的均摊时间复杂度均为 $O(1)$，但前插和删除复杂度为 $O(n)$。

所以，当你需要一个序列时，如果你多半与头部元素打交道，主要使用 List；如果你需要访问一般元素，则使用 Vector。Vector 是一个强大、通用的，具有优异全能表现的集合。不过，在有些情况下，ArrayBuffer 能提供更低的常数时间，从而降低开销，提高性能。

其他的通用场景中，我们需要基于键的、$O(1)$ 复杂度的元素存取，也就是数值根据键被存储在 immutable.Map（mutable.Map）中。同样，immutable.Set（mutable.Set）被用来测试值是否存在。

## 12.3 集合库的设计惯例

为了解决设计问题，促进代码重用，集合库使用了许多惯用方法。接下来，我们会讨论这些惯用法，同时了解库实现中用到的“辅助”类型。

### 12.3.1 Builder

我在前面曾提到，如果小心合理地使用，可变集合将是为了提高性能而做出的合理让步。事实上，集合 API 在内部使用了可变集合建立新的输出集合，如在 `map` 操作中就是如此。`map` 操作还使用了 `collection.mutable.Builder` 特征的实现，以构造新实例。

生成器（builder）的签名如下：

```
trait Builder[-Elem, +To] {
  def +=(elem: Elem): Builder.this.type
  def clear()
  def result(): To
  ...  // Other methods derived from these three abstract methods.
}
```

这种罕见的 `Builder.this.type` 签名是一个*单例类型*（singleton type）。它确保 `+=` 方法在调用时只返回生 `Builder` 的实例，也就是 `this`。如果尝试返回 `Builder` 的一个新实例，那么就无法通过类型检查！我们将会在 15.3 节中学习到单例。

以下是 `List` 的一个 `builder` 实现：

```
// src/main/scala/progscala2/collections/ListBuilder.sc
import collection.mutable.Builder

class ListBuilder[T] extends Builder[T, List[T]] {

  private var storage = Vector.empty[T]

  def +=(elem: T) = {
    storage = storage :+ elem
    this
  }

  def clear(): Unit = { storage = Vector.empty[T] }

  def result(): List[T] = storage.toList
}

val lb = new ListBuilder[Int]
(1 to 3) foreach (i => lb += i)
lb.result
// 结果: List(1, 2, 3)
```

一个比 `Vector` 更有效率的内部存储集合诞生了，不过它说明了 `builder` 的用法。

## 12.3.2 CanBuildFrom

考虑如下示例，对列表中的数字做 map 操作：

```
scala> List(1, 2, 3, 4, 5) map (2 * _)
res0: List[Int] = List(2, 4, 6, 8, 10)
```

List 中有这种方法，它的简化签名为：

```
map[B](f: (A) => B): List[B]
```

然而，标准库尽可能做到重用。回想一下，5.2.3 节提到：map 事实上定义于 scala.collection.TraversableLike，这是一个被 List 混入的 trait。map 的实际签名如下：

```
trait TraversableLike[+A, +Repr] extends ... {
  ...
  def map[B, That](f: A => B)(
    implicit bf: CanBuildFrom[Repr, B, That]): That = {...}
}
```

Repr 是内部用来保存元素的集合类型。B 是函数 f 创建的元素类型。That 是我们想要创建的目标集合的类型参数，它可能与输入的原始集合相同，也可能不同。

TraversableLike 对其子类（如 List）一无所知，但它还是可以构造新的 List 并将其返回，因为隐含的 CanBuildFrom 实例封装了所需要的细节。

CanBuildFrom 是创建 Builder 实例的工厂的 trait，由工厂来完成实际构造新集合的工作。

使用 CanBuildFrom 技术的一个缺点是额外增加了方法签名的复杂度。然而，除了促进类似 map 等操作在面向对象中的重用以外，CanBuildFrom 在其他方面使得构造更加模块化、通用化。

例如：CanBuildFrom 实例可以为返回的不同具体集合实例化多个 Builder。通常它会返回相同类型的新集合，或者返回对给定元素来说更高效的子类型。

实现一个包含很多元素的映射表最好将键存储在散列表中，同时提供均为 $O(1)$ 的存储和查询复杂度。然而，对于一个小映射，可能将元素直接存在数组或列表中会更快，这时 $n$ 很小，$O(n)$ 的查询复杂度事实上比散列表的 $O(1)$ 复杂度还要快，这取决于散列表的常数开销因子。

输入集合类型不能用来作为输出集合类型的情况还有其他形式。考虑下面的示例：

```
scala> val set = collection.BitSet(1, 2, 3, 4, 5)
set: scala.collection.BitSet = BitSet(1, 2, 3, 4, 5)

scala> set map (_.toString)
res0: scala.collection.SortedSet[String] = TreeSet(1, 2, 3, 4, 5)
```

BitSet（http://www.scala-lang.org/api/current/#scala.collection.BitSet）只能保存整数，所以，如果将其元素转为字符串，隐含的 CanBuildFrom 只能实例化不同于输入的输出集合。在本例中，输出集合是 SortedSet（http://www.scala-lang.org/api/current/#scala.collection.immutable.SortedSet）。

类似地，对于字符串（字符序列），我们也可能遇到以下情形：

```
scala> "xyz" map (_.toInt)
res0: scala.collection.immutable.IndexedSeq[Int] = Vector(120, 121, 122)
```

CanBuildFrom 的另一个好处是传送其他上下文信息的能力，这些上下文信息可能是原集合不知道或不适合由原集合来传递的。例如：使用分布式计算 API 时，特定的 CanBuildFrom 实例可能被用于构建那些适合序列化到远程进程的集合。

### 12.3.3 Like特征

我们看到 Builder 和 CanBuildFrom 为输出集合指定了类型参数。为了支持对这些参数的指定，也为了加强代码重用，你知道的大多数集合都混入了相应的…Like 特征。该 trait 可以添加合适的返回值类型，并为常见方法提供实现。

以下是 collection.immutable.Seq 的声明：

```
trait Seq[+A] extends Iterable[A] with collection.Seq[A]
 with GenericTraversableTemplate[A, Seq] with SeqLike[A, Seq[A]]
 with Parallelizable[A, ParSeq[A]]
```

需要注意的是，collection.SeqLike 同时用元素类型 A 和 Seq[A] 本身进行参数化。第二个参数用于约束在 map 等方法中能够使用的 CanBuildFrom 实例。该 trait 还实现了 Seq 中大部分我们熟悉的方法。

我鼓励大家查阅 Scaladoc 中的 collection.immutable.Seq 和我们讨论的其他公共集合类型。点击其他 trait 的链接，看看它们都做了什么。这些 trait 及其自身混入的 trait 构成了一个类型树。幸运的是，对于实际使用具体的集合类型，这里的大部分细节都是无关紧要的。

总之，在集合的设计中使用了三个最重要的设计方法。

(1) 用 Builder 抽象了构造。
(2) 用 CanBuildFrom 提供隐含的工厂，用于构造适合给定上下文的生成器实例。
(3) Like 特征为 Builder 和 CanBuildFrom 添加必须的返回值类型参数，并提供大部分的方法实现。

如果要构建自己的集合，你需要遵循这些惯例。根据第 7 章，我们知道，如果集合实现了 foreach、map、flatMap 和 withFilter，那么集合就可以像内置的集合类型一样在 for 表达式中使用。

## 12.4 值类型的特化

Scala 对值类型（如 Int、Float 等）和引用类型进行统一处理的一个好处就是，Scala 可以用值类型声明参数化类型实例，如 List[Int]。与此相反，Java 中只能用包装过的类型，如 List<Integer>。包装需为对象分配额外的内存，也需要额外的时间管理内存。而原生类型在内存中是连续分布的，可以提供缓存命中率，从而为某些算法提高性能。

因此，在以数据为中心的 Java 库中（如那些为大数据应用写的库中），往往有一长串为原

生类型专门定义的容器声明，也可能只有几个（为 long 和 double 定义的容器）。也就是说，你会看到有的类表示 long 型的向量，有的类表示 double 型的向量。于是，库的体积远大于 Java 支持原生类型的集合的情况，而这些专门声明的原生类型的容器，往往效率比相应的基于对象的实现高出十倍。

不幸的是，尽管 Scala 允许我们声明值类型的容器，但这并不能解决问题。由于类型擦除的存在以及 JVM 很难记忆容器中元素类型的事实，元素被认为是 Object 类型，所有的元素类型共有同一种容器的实现版本。所以 List[Double] 依然需要使用例如 Java 包装后的 Double 样的类型。

如果存在一种机制能告诉编译器生成容器的“特化”实现，该有多好？容器的这种“特化”实现有助于原型类型的最优化。事实上，Scala 中的 @specialized（http://www.scala-lang.org/api/current/scala/specialized.html）标记可以实现这个目的。它会告诉编译器为标记中列出的值类型生成自定义的实现：

```
class SpecialVector[@specialized(Int, Double, Boolean) T] {...}
```

这个例子为 Int、Double 和 Boolean 产生特化版本的 SpecialVector。如果省略掉 @specialized 后的列表，所有值类型都将产生特化的版本。

然而，自从引入 @specialized，实践中就暴露出了它的一些限制。首先，它会引入很多自动生成的代码，所以滥用 @specialize 将会使库的体积过大。

第二，实现中存在一些设计缺陷（关于本问题的详细讨论，https://speakerdeck.com/vladureche/miniboxing-presentation-at-scaladays-2014 有最新更新）。如果一个字段在原始容器中被声明为泛型类型，那么它在特化时不会转为原生类型的字段。相反，编译器会再生成一个相应原生类型的重复字段，从而带来错误。另一个缺陷是，特化的容器被实现为原始通用容器的子类。当通用容器及其子类都是特化类型时，就无法工作了。特化的类型应该与父类型具有同样的继承关系，但由于 JVM 的单继承模型，这种相同的继承关系得不到支持。

由于这些实践中的不足，Scala 库对 @specialized 使用得很有限。它主要用在 FunctionN、TupleN、ProductN 类型或者少数集合中。

在我们讨论 @specialized 的一个新兴替代品之前，要注意到，方法还有一个 @unspecialized（http://www.scala-lang.org/api/current/index.html#scala.annotation.unspecialized）标记。当类型有 @specialized 标记，但又不想使用生成特化版本的方法时，我们可以使用 @unspecialized 标记。当特化带来的性能改进优势不足以弥补它带来的库体积增大时，你也可以使用这个标记。

## Miniboxing

Miniboxing 试图去掉特化的那些限制因素，因此它是 @specialized 的一个替代机制，目前正处在开发中。尽管做为编译器插件（http://scala-miniboxing.org）已经可以用于实验了，Miniboxing 在 Scala 的下一个版本中可能才会出现。所以，现在可以讨论一下它了。

安装了这个插件后，它的使用方法与 @specialized 是一致的：

```
class SpecialVector[@miniboxed(Int, Double, Boolean) T] {...}
```

Miniboxing 将泛型容器改为拥有两个子类型的 trait，一个子类用于值类型，另一个子类用于引用类型，从而降低了代码膨胀。表示原生类型的子类注意到，一个 8 比特的值可以装下任何一个原生类型。利用这一点，该类增加了一个“标签”，用于表示这 8 比特应被解释成的原生类型。所以，该子类表现得像一个带标签的联合体，不再需要让每个原生类型都拥有一个单独的实例。引用类型不受影响。

通过将原始容器改为 trait，它所存在的所有继承关系都得到了保持。例如：我们有两个参数化的容器 C[T] 和 D[T]，其中 D[T] 类继承了 C[T]，并且这两个类型都做了特化，那么从概念上看，生成的代码会是这样：

```
trait C[T]                           // was class C[T]
class C_primitive[T] extends C[T]    // T is an AnyVal
class C_anyref[T] extends C[T]       // T is an AnyRef

trait D[T] extends C[T]                                 // was class D[T]
class D_primitive[T] extends C_primitive[T] with D[T]   // T is an AnyVal
class D_anyref[T] extends C_anyref[T] with D[T]         // T is an AnyRef
```

同时，为了提高效率，你依然可以使用 @specialized 标记。但是需要注意它会引起库体积变大，以及前文介绍过的设计上的限制问题。

## 12.5 本章回顾与下一章提要

我们对 Scala 的集合库进行了深入了解，包括可变与不可变的区别，并行变体，以及如何在 Java 集合和 Scala 集合间相互转换；还讨论了一个重要的未结束的话题，即：如何使集合高效地与 JVM 原生类型配合，以减少对原生类型进行包装带来的开销。

在涉及 Scala 类型系统的主要话题之前，下一章将涵盖的话题你也应该了解。尽管这个话题不是每天都关注的：Scala 对细粒度可见性控制的各种支持。Scala 远远超过了 Java 中的 public、protected、private 可见性控制和默认的 package 的作用域管理。

第 13 章

# 可见性规则

Java 提供了四个级别的可见性：public、protected、private 以及默认的 package 可见性。而 Scala 的可见性规则很好地超越了这四个级别。在面向对象的编程语言中，可见性规则用于对外暴露某一类型必须提供的公有抽象，而实现信息则被封装起来，外部无法访问。

本章将对可见性规则进行深入讲解，而这部分内容确实会有些枯燥。从表面上看，除了 Scala 的默认可见性是 public 之外，Scala 的可见性规则与 Java 规则非常相似。Scala 增加了许多复杂的作用域规则，不过在日常编写的 Scala 代码中，你并没有太多机会应用到这些规则。因此，你可以考虑粗略浏览本章前一两个小节以及总结部分。其他内容可以暂时不看，等到你开始编写自己的 Scala 库，需要了解特定的内容时，再查阅相关内容。

## 13.1 默认可见性：公有可见性

与 Java 不同，Scala 将公有可见性（public visibility）视为默认可见性，不过这与其他的面向对象编程语言的做法一致。如果你希望对类型中某些成员的可见性进行限制，只允许类型本身访问这些成员，常用的做法是将这些类型成员声明为私有成员，或者把它们声明为 protected 成员，只允许子类访问。不过 Scala 为你提供了更多选项，而本章会做出详细地讲述。

对于用户需要查看并使用的任何对象成员，你都可以把它们设置 public 可见性。在使用 public 可见性时，请牢记一点：这组具有公有可见性的成员组合成了一个抽象体，与类型名称一起暴露给了外部。

> 定义最小化、清晰而又简洁的公有抽象也是良好的面向对象设计中的一环。

传统的面向对象设计认为，字段应该是 private 或 protected 可见性。假如需要对该字段进行访问，我们应该提供访问方法，而不是把所有的字段默认为可访问。

存在两个理由支撑这一传统。第一个理由是：这样做能够阻止用户未经许可便对可变变量进行修改。不过，使用不可变值也可以消除这一顾虑。而第二个理由是：字段只是实现的一部分，它们并不是你希望对外暴露的公有抽象体。

如果允许直接访问字段，那么统一访问原则（请参考 8.6.1 节的相关内容）的优势便体现出来了。运用统一访问原则，我们可以让用户使用公用字段访问语法对字段进行访问，而在实现时可以根据情况，采用方法访问或直接访问的方式。用户则无需知道具体的实现方式。尽管仍然需要重新编译代码，但我们可以在不告知用户的情况下修改具体的实现。

使用类型的“用户”存在两种：继承自该类型的类型，以及使用该类型实例的代码。与操作实例的用户相比，继承类型往往需要对父类型成员进行更多次的访问。

Scala 可见性规则与 Java 规则相似，不过 Scala 规则更统一，也更灵活。例如：在 Java 中，如果一个内部类中包含了私有成员，那么包装类能够访问该成员。在 Scala 中，包装类无法看到内部类的私有成员，不过 Scala 提供了另外一种声明方式，使得包装类可以看到该成员。

## 13.2 可见性关键字

在 Java 和 C# 语言中，我们在成员声明起始位置输入用于修改可见性的关键字，如 `private` 和 `protected` 关键字。在 Scala 中，你会在类型的 `class` 或 `trait` 关键字之前、字段的 `val` 或 `var` 之前、方法定义的 `def` 关键字之前找到这些可见性关键字。

> 你也可以在类的主构造函数中使用访问修饰符。把这些修饰符放到类型名称和类型参数（如果声明中有类型参数的话）之后，参数列表之前。相关示例如下：`class Restricted[+A] private (name: String) {...}`。
>
> 为什么要限制用户访问主构造函数呢？因为这样一来，便能强迫用户调用工厂方法，而不是直接初始化该类型。

表 13-1 对可见性作用域进行了总结。

**表13-1：可见性作用域**

| 可见性名称 | 关键字 | 描　　述 |
|---|---|---|
| public | 无关键字 | 任何作用域内均能访问公有成员或公有类型，public 可见性无视所有限制 |
| protected | `protected` | 受保护（protected）成员对本类型、继承类型以及嵌套类型可见。而受保护的类型则只对相同包内或子包内的某些类型可见 |
| private | `private` | 私有（private）成员只对本类型和嵌套类型可见，而且可以访问私有成员的类型必须位于相同包内 |

（续）

| 可见性名称 | 关键字 | 描　　述 |
|---|---|---|
| 作用域内受保护 | protected[scope] | 只能在某一作用域内可见，该作用域可以是包作用域、类型作用域或 this 作用域（这意味着如果成员使用了该可见性，this 代表了相同实例，如果类型使用该可见性，则 this 表示该类型所在的包）。如果想了解更多细节，请阅读下面的文字 |
| 作用域内私有 | private[scope] | 除了对具有继承关系的类的可见性不同之外（会在后续说明），与作用域内受保护可见性相同 |

我们将会对这些可见性选项进行更深入地研究。为了简化起见，我们在示例中将以成员字段为例。方法声明和类型声明中的可见性规则与字段规则一致。

> 不幸的是，你无法为包添加任何可见性修改符。因此，包总是公有的，即便是包里面包含了一些非公有可见的类型。

## 13.3　Public可见性

对于任何声明体，如果未指定可见性，它们对应的可见性关键字便默认为 public，这意味着在任何作用域中，该声明体均可见。Scala 未提供 public 关键字，这与 Java 截然不同。Java 语言中默认的“公有”可见性只对包可见（即包内私有）。而其他的面向对象语言（如 Ruby），同样将 public 作为默认作用域：

```
// src/main/scala/progscala2/visibility/public.scala

package scopeA {
  class PublicClass1 {
    val publicField = 1

    class Nested {
      val nestedField = 1
    }

    val nested = new Nested
  }

  class PublicClass2 extends PublicClass1 {
    val field2  = publicField + 1
    val nField2 = new Nested().nestedField
  }
}

package scopeB {
  class PublicClass1B extends scopeA.PublicClass1

  class UsingClass(val publicClass: scopeA.PublicClass1) {
    def method = "UsingClass:" +
```

```
        " field: " + publicClass.publicField +
        " nested field: " + publicClass.nested.nestedField
    }
  }
```

代码中出现的包和类都是公有可见性。请注意，scopeB.UsingClass 类能访问 scopeA.PublicClass1 类及其成员，包括 Nested 类的实例及其公有字段。

## 13.4 Protected可见性

继承类往往需要对父类型的一些详细信息进行访问，而 protected 可见性则为这些继承类型的开发者提供了便利。所有使用 protected 关键字声明的成员只对该定义类型可见，包括相同类型的其他实例以及所有的继承类型。如果某一类型被声明为受保护类型，那么该类型只对包含这个类型的包内可见。

Java 则恰恰相反，包内所有对象均可访问 protected 成员。Scala 则通过设置作用域内私有和受保护访问控制处理这种场景：

```
// src/main/scala/progscala2/visibility/protected.scalaX
// 无法通过编译

package scopeA {
  class ProtectedClass1(protected val protectedField1: Int) {
    protected val protectedField2 = 1

    def equalFields(other: ProtectedClass1) =
      (protectedField1 == other.protectedField1) &&
      (protectedField2 == other.protectedField2) &&
      (nested == other.nested)

    class Nested {
      protected val nestedField = 1
    }

    protected val nested = new Nested
  }

  class ProtectedClass2 extends ProtectedClass1(1) {
    val field1 = protectedField1
    val field2 = protectedField2
    val nField = new Nested().nestedField  // 错误
  }

  class ProtectedClass3 {
    val protectedClass1 = new ProtectedClass1(1)
    val protectedField1 = protectedClass1.protectedField1 // 错误
    val protectedField2 = protectedClass1.protectedField2 // 错误
    val protectedNField = protectedClass1.nested.nestedField // 错误
  }

  protected class ProtectedClass4
```

```
  class ProtectedClass5 extends ProtectedClass4
  protected class ProtectedClass6 extends ProtectedClass4
}

package scopeB {
  class ProtectedClass4B extends scopeA.ProtectedClass4 // 错误
}
```

运行 scalac 对代码文件进行编译时，将会出现下面列出的五个错误，分别对应了使用“// 错误”注释的五行代码：

```
.../visibility/protected.scalaX:23: error: value nestedField in class
Nested cannot be accessed in ProtectedClass2.this.Nested
 Access to protected value nestedField not permitted because
 enclosing class ProtectedClass2 in package scopeA is not a subclass of
 class Nested in class ProtectedClass1 where target is defined
    val nField = new Nested().nestedField  // 错误
                             ^
...
5 errors found
```

由于 ProtectedClass2 继承了 Protected1 类，因此 ProtectedClass2 能访问 ProtectedClass1 中定义的受保护成员。不过，ProtectedClass2 无法访问 protectedClass1.nested 对象中受保护的 nestedField 成员。同时，ProtectedClass3 类也无法访问它使用的 ProtectedClass1 实例中的受保护成员。

最后，由于 ProtectedClass4 被声明为 protected 类，其对 scopeB 包内的对象不可见。

# 13.5　Private可见性

私有（private）可见性将实现细节完全隐藏起来，即便是继承类的实现者也无法访问这些细节。声明中包含了 private 关键字的所有成员都只对定义该成员的类型可见，该类型的其他实例也能访问这些成员。如果类型被声明为私有可见性类型，那么该类型的可见性将被限定到包含该类型的包内：

```
// src/main/scala/progscala2/visibility/private.scalaX
// 无法通过编译

package scopeA {
  class PrivateClass1(private val privateField1: Int) {
    private val privateField2 = 1

    def equalFields(other: PrivateClass1) =
      (privateField1 == other.privateField1) &&
      (privateField2 == other.privateField2) &&
      (nested == other.nested)

    class Nested {
      private val nestedField = 1
    }

    private val nested = new Nested
```

```
  }

  class PrivateClass2 extends PrivateClass1(1) {
    val field1 = privateField1  // 错误
    val field2 = privateField2  // 错误
    val nField = new Nested().nestedField // 错误
  }

  class PrivateClass3 {
    val privateClass1 = new PrivateClass1(1)
    val privateField1 = privateClass1.privateField1 // 错误
    val privateField2 = privateClass1.privateField2 // 错误
    val privateNField = privateClass1.nested.nestedField // 错误
  }

  private class PrivateClass4

  class PrivateClass5 extends PrivateClass4  // 错误
  protected class PrivateClass6 extends PrivateClass4 // 错误
  private class PrivateClass7 extends PrivateClass4
}

package scopeB {
  class PrivateClass4B extends scopeA.PrivateClass4  // 错误
}
```

编译上述文件时，注释中包含了“错误”注释的九行代码将出现错误。

现在，PrivateClass2 无法访问父类 PrivateClass1 的私有成员。正如错误信息表示的那样，这些私有成员对子类完全不可见。而嵌套类中的私有字段也是无法访问的。

与访问受保护变量的示例一样，PrivateClass3 正在对 PrivateClass1 实例进行操作，但它却无法访问该实例的私有成员。请注意，equalFields 方法可以访问其他实例中定义的私有成员。

privateClass5 和 privateClass6 声明之所以失败，是因为假如 Scala 允许出现这样两个类的话，PrivateClass4 便“逃脱了 private 作用域的限定”。不过，由于 PrivateClass7 也被声明为私有类，因此 PrivateClass7 声明成功。奇怪的是，在之前的示例中，我们声明了一个继承自受保护类的公有类，但却未出现这样的错误。

最后，与受保护的类型声明体一样，我们无法在不同的包中定义该私有类型的子类。

## 13.6 作用域内私有和作用域内受保护可见性

Scala 为用户提供了一些额外方法，以帮助用户以更小的粒度对可见性的作用域进行调整。从这一点看，Scala 超过了大多数的语言。Scala 提供了作用域内私有（scoped private）可见性声明和作用域内受保护（scoped protected）可见性声明。请注意，在具有继承关系的情况下，对类成员应用这两类可见性后表现不同。但除此之外，这两类可见性的表现完全一致，因此在同一个作用域内，私有可见性可以和受保护可见性交换使用。

> 尽管这两类可见性规则几乎一样，不过在 Scala 库中 `private[X]` 的出现次数略多于 `protected[X]` 的出现次数。在本书的第 1 版中，我们注意到 Scala 2.7.X 版本中 `private[X]` 的出现次数比 `protected[X]` 的出现次数多了五次。而在 Scala 2.11 中，两者的比例更为接近，它们的出现比例达到了 5/3。

我们首先对作用域内私有可见性和作用域内受保护可见性之间的差异部分进行思考。在具有继承关系的情况下，如果某些成员分别具有这两类可见性，那么这两类可见性的表现如何呢？

```
// src/main/scala/progscala2/visibility/scope-inheritance.scalaX
// 无法通过编译

package scopeA {
  class Class1 {
    private[scopeA]   val scopeA_privateField = 1
    protected[scopeA] val scopeA_protectedField = 2
    private[Class1]   val class1_privateField = 3
    protected[Class1] val class1_protectedField = 4
    private[this]     val this_privateField = 5
    protected[this]   val this_protectedField = 6
  }

  class Class2 extends Class1 {
    val field1 = scopeA_privateField
    val field2 = scopeA_protectedField
    val field3 = class1_privateField     // 错误
    val field4 = class1_protectedField
    val field5 = this_privateField       // 错误
    val field6 = this_protectedField
  }
}

package scopeB {
  class Class2B extends scopeA.Class1 {
    val field1 = scopeA_privateField     // 错误
    val field2 = scopeA_protectedField
    val field3 = class1_privateField     // 错误
    val field4 = class1_protectedField
    val field5 = this_privateField       // 错误
    val field6 = this_protectedField
  }
}
```

这个文件将产生五处编译错误。

最开始的两个错误位于 `Class2` 内，这两个错误告诉我们，在相同包内定义的继承类不能引用父类的作用域内私有成员，也不能引用父类中定义的 `this` 作用域内私有成员。不过继承类能够引用限定在包裹了 `Class1` 和 `Class2` 的包（或者类型）作用域内的私有成员。

相比之下，在 `Class1` 所在包外部定义的继承类则无法访问 `Class1` 中定义的任何作用域内私有成员。

不过，作用域内受保护成员对其继承类 Class2 和 Class2B 均可见。

由于在 Scala 库中受限私有可见性出现的次数略多过受限受保护可见性出现的次数，因此除了那些像之前出现的继承关系会对可见性造成影响的情况之外，我们将在后续的例子和讨论中只使用作用域内私有可见性。

由于 private[this] 可见性能够对类型成员造成影响，因此它也是最受限制的可见性，我们也将首先对其进行讲解：

```
// src/main/scala/progscala2/visibility/private-this.scalaX
// 无法通过编译

package scopeA {
  class PrivateClass1(private[this] val privateField1: Int) {
    private[this] val privateField2 = 1

    def equalFields(other: PrivateClass1) =
      (privateField1 == other.privateField1) && // 错误
      (privateField2 == other.privateField2) && // 错误
      (nested == other.nested) // 错误

    class Nested {
      private[this] val nestedField = 1
    }

    private[this] val nested = new Nested
  }

  class PrivateClass2 extends PrivateClass1(1) {
    val field1 = privateField1  // 错误
    val field2 = privateField2  // 错误
    val nField = new Nested().nestedField  // 错误
  }

  class PrivateClass3 {
    val privateClass1 = new PrivateClass1(1)
    val privateField1 = privateClass1.privateField1  // 错误
    val privateField2 = privateClass1.privateField2  // 错误
    val privateNField = privateClass1.nested.nestedField // 错误
  }
}
```

编译这段代码时，编译器将报出九处错误。

由于第 10 行代码和第 11 行代码都是始于第 9 行表达式的一部分，而编译器在第 9 行发现错误后便会停止对该表达式进行解析，因此第 10 行和第 11 行不会被解析。

private[this] 成员仅对相同实例可见。同一类型的某一实例无法访问其他实例的 private[this] 成员，因此 equalFields 方法无法通过解析。

除此之外，使用 private[this] 修饰的类成员的可见性与未指定作用域范围的 private 可见性一致。

使用 private[this] 声明类型时，this 也只能作用于包含该类型的包中，如下所示：

```
// src/main/scala/progscala2/visibility/private-this-pkg.scalaX
// 无法通过编译

package scopeA {
  private[this] class PrivateClass1

  package scopeA2 {
    private[this] class PrivateClass2
  }

  class PrivateClass3 extends PrivateClass1  // 错误
  protected class PrivateClass4 extends PrivateClass1 // 错误
  private class PrivateClass5 extends PrivateClass1
  private[this] class PrivateClass6 extends PrivateClass1

  private[this] class PrivateClass7 extends scopeA2.PrivateClass2 // 错误
}

package scopeB {
  class PrivateClass1B extends scopeA.PrivateClass1 // 错误
}
```

这段代码将产生四处错误。

在相同包中，无法成功地为一个 private[this] 类型声明 public 或 protected 子类，你只能为其声明 private 和 private[this] 子类。与此同时，由于 PrivateClass2 的可见性被限定在 scopeA2 作用域内，因此你无法在 scopeA2 作用域外声明其子类。同理，在与 scopeA2 无关的 scopeB 作用域内使用 PrivateClass1 声明类同样会失败。

由此，我们可以给出结论：如果使用 private[this] 修饰类型，那么此时的 private[this] 等同于 Java 语言中的包内私有可见性。

下面，我们将对类型级别可见性进行分析，我们将分析 private[T] 可见性，其中 T 代表了某一类型：

```
// src/main/scala/progscala2/visibility/private-type.scalaX
// 无法通过编译

package scopeA {
  class PrivateClass1(private[PrivateClass1] val privateField1: Int) {
    private[PrivateClass1] val privateField2 = 1

    def equalFields(other: PrivateClass1) =
      (privateField1 == other.privateField1) &&
      (privateField2 == other.privateField2) &&
      (nested  == other.nested)

    class Nested {
      private[Nested] val nestedField = 1
    }

    private[PrivateClass1] val nested = new Nested
    val nestedNested = nested.nestedField   // 错误
  }
```

```
    class PrivateClass2 extends PrivateClass1(1) {
      val field1 = privateField1  // 错误
      val field2 = privateField2  // 错误
      val nField = new Nested().nestedField  // 错误
    }

    class PrivateClass3 {
      val privateClass1 = new PrivateClass1(1)
      val privateField1 = privateClass1.privateField1  // 错误
      val privateField2 = privateClass1.privateField2  // 错误
      val privateNField = privateClass1.nested.nestedField // 错误
    }
  }
```

该文件中包含了七处访问错误。

由于可见性类型为 private[PrivateClass1] 的成员对其他同类型实例可见，因此 equalFields 能够通过解析。所以可以给出结论：private[T] 没有 private[this] 这样严格。请留意，因为 Nested.nestedField 被声明为 private[Nested] 可见性，所以 PrivateClass1 无法访问该字段。

> 假如 T 类型的某些成员被声明为 private[T] 成员，那么该成员的行为与使用 private 修饰的行为一致。private[T] 并不等同于 private[this]，后者限制更为严格。

假如我们将 Nested.nestedField 的作用域修改为 private[PrivateClass1]，会发生什么呢？下面我们将讨论 private[T] 会如何对嵌套类型造成影响：

```
// src/main/scala/progscala2/visibility/private-type-nested.scalaX
// 无法通过编译

package scopeA {
  class PrivateClass1 {
    classNested {
      private[PrivateClass1] val nestedField = 1
    }

    private[PrivateClass1] val nested = new Nested
    val nestedNested = nested.nestedField
  }

  classPrivateClass2 extends PrivateClass1 {
    val nField = new Nested().nestedField   // 错误
  }

  class PrivateClass3 {
    val privateClass1 = new PrivateClass1
    val privateNField = privateClass1.nested.nestedField // 错误
  }
}
```

执行这段代码将会出现两处错误。

尽管现在 nestedField 对 PrivateClass1 可见，但它仍然对 PrivateClass1 外部的类型不可见。这与 Java 中 private 修饰词的作用一致。

下面我们将使用包名对作用域进行限制：

```
// src/main/scala/progscala2/visibility/private-pkg-type.scalaX
// 无法通过编译

package scopeA {
  private[scopeA] class PrivateClass1

  package scopeA2 {
    private [scopeA2] class PrivateClass2
    private [scopeA]  class PrivateClass3
  }

  class PrivateClass4 extends PrivateClass1
  protected class PrivateClass5 extends PrivateClass1
  private class PrivateClass6 extends PrivateClass1
  private[this] class PrivateClass7 extends PrivateClass1

  private[this] class PrivateClass8 extends scopeA2.PrivateClass2 // 错误
  private[this] class PrivateClass9 extends scopeA2.PrivateClass3
}

package scopeB {
  class PrivateClass1B extends scopeA.PrivateClass1 // 错误
}
```

编译该文件时也会产生两个错误。

请留意，现在我们无法在 scopeA2 作用域外将 PrivateClass2 子类化。不过由于 PrivateClass3 被声明为 private[ScopeA] 类型，因此我们可以在 scopeA 作用域内能将 PrivateClass3 子类化。

最后，我们再看看对类型参数施加包级作用域可见性限定后，会产生什么影响：

```
// src/main/scala/progscala2/visibility/private-pkg.scalaX
// 无法通过编译

package scopeA {
  class PrivateClass1 {
    private[scopeA] val privateField = 1

    class Nested {
      private[scopeA] val nestedField = 1
    }

    private[scopeA] val nested = new Nested
  }

  class PrivateClass2 extends PrivateClass1 {
    val field  = privateField
```

```
    val nField = new Nested().nestedField
  }

  class PrivateClass3 {
    val privateClass1 = new PrivateClass1
    val privateField  = privateClass1.privateField
    val privateNField = privateClass1.nested.nestedField
  }

  package scopeA2 {
    class PrivateClass4 {
      private[scopeA2] val field1 = 1
      private[scopeA]  val field2 = 2
    }
  }

  class PrivateClass5 {
    val privateClass4 = new scopeA2.PrivateClass4
    val field1 = privateClass4.field1  // 错误
    val field2 = privateClass4.field2
  }
}

package scopeB {
  class PrivateClass1B extends scopeA.PrivateClass1 {
    val field1 = privateField   // 错误
    val privateClass1 = new scopeA.PrivateClass1
    val field2 = privateClass1.privateField  // 错误
  }
}
```

该文件有三处错误。

如果我们试图从某个与 scopeA 无关的包 scopeB 中访问 scopeA 时，或者当我们尝试从嵌套包 scopeA2 中访问成员变量时，便会出现错误。而这便是上面代码的所有错误来源。

> 假如某一类型或成员被声明为 private[P] 可见性，P 是该类型或成员所在的包，那么该可见性等同于 Java 中的包内私有可见性。

## 13.7　对可见性的想法

Scala 的可见性声明非常灵活，而且这些声明表现一致。从实例级（private[this]）到包级（private[P]，P 为对应的包名），这些可见性规则在各种作用域里均提供了细粒度控制约束。例如：使用这些可见性声明能够很容易地创建可重用组件，这些组件的类型对外暴露给了组件最顶层包之外的代码，但实现类型和类型成员隐藏在组件包中。

尽管标准库之外的代码中并未广泛使用这些细粒度的可见性约束，但它们理应得到广泛使用。假如你正在编写自己的 Scala 库，那么请思考一下，哪些类型和方法不应对客户开放，

之后请为这些类型和方法添加合适的可见性规则。

最后，我们发现了一个关于 trait 中隐藏成员的潜在问题，请查看下面的提示。

> 为 trait 成员选择名字时，请保持谨慎。假如两个 trait 中某一成员名字相同，而某一实例同时使用了这两个 trait，那么即使这两个重名的成员都是私有成员，还是会导致命名冲突。

幸运的是，编译器会捕获命名冲突这一问题。

## 13.8 本章回顾与下一章提要

通过使用 Scala 提供的可见性规则，用户可以以较细的粒度对功能的可见性进行控制。你也可以什么都不做，直接使用默认的公有可见性。不过留意哪些功能应对外暴露也是好的类库设计的一部分。而在类库内，对各个组件之间的可见性进行限定能有助于确保类库的健壮性，并能使长期维护更加简单。

现在，我们即将开启一段关于 Scala 类型系统的学习旅程。我们已经了解了很多类型系统的相关知识，不过为了彻底挖掘出类型系统的强大威力，我们还需要系统地理解类型系统。

## 第 14 章

# Scala类型系统（I）

Scala 是一种静态类型语言。它的类型系统可以说是所有编程语言中最复杂的，一部分原因是它将函数式编程和面向对象编程的思想结合了起来。它的类型系统力求逻辑完备、完整、一致，并修复了 Java 类型系统的一些限制。

理想情况下，类型系统应该具有足够的表达能力，来防止程序处于无效状态。它将在编译时加强限制，使得运行时的失败不会发生。在实践中，我们离这一目标非常远，但 Scala 类型系统向这个长期目标又迈进了一步。

不过，Scala 的类型系统可能一开始看起来很吓人。这是 Scala 语言中最有争议的特性。当人们声称，Scala 很“复杂”时，他们通常指的就是其类型系统。

幸运的是，类型推断为我们隐藏了很多细节。尽管最终你需要熟悉类型系统的大部分结构，但对 Scala 的运用并不需要精通类型系统。

对类型系统我们已经了解了很多。本章会把之前的知识结合起来，介绍每一个 Scala 初学者都应该了解的常用特性。下一章介绍的更高级特性则没这么重要，Scala 初学者不需要立刻就开始学习。在第 24 章，我们将讨论反射（运行时内省）和宏的中的类型信息。

我们还将讨论 Scala 类型系统与 Java 类型系统的相似之处，这对你来说可能比较熟悉。理解这些差异对 Scala 与 Java 库的互操作非常有用。

事实上，Scala 的类型系统之所以复杂，部分原因在于 Scala 需要拥有 Java 类型系统的特性以支持与 Java 的互操作。

下面我们将从熟悉的参数化类型开始。

## 14.1 参数化类型

我们已经在好多场合遇到过参数化类型了。在 2.13 节，我们将其与抽象类型相对比。在 10.1 节中，我们又探讨了子类派生时的变异。

在本节中，我们将回顾一下上述的部分细节，然后再补充一些我们应该了解的其他知识。

### 14.1.1 变异标记

首先，让我们回忆一下变异标记是如何工作的。声明 `List[+A]` 表示 `List` 被类型 `A` 参数化了。`+` 是变异声明，在这里表示 `List` 对类型参数是协变的。协变意味着 `List[String]` 被认为是 `List[AnyRef]` 的子类型，因为 `String` 是 `AnyRef` 的子类型。

同样，`-` 变异标记表示该类型对其类型参数是逆变的。`FunctionN` 函数传入的 `N` 个参数就是其中一个例子。比如：`Function2` 的类型签名为 `Function2[-T1, -T2, +R]`。我们在 10.1.1 节中解释了函数参数的类型必须是逆变的原因。

### 14.1.2 类型构造器

有时候，你会在参数化类型中遇到*类型构造器*（type constructor）这一术语。它反映了参数化类型创建特定类型的方式，这与用于生成类的实例构造器大致类似。

例如：`List` 是 `List[String]` 和 `List[Int]` 的类型构造器，通过 `List` 构造了两个不同的类型。事实上，你可以说，所有的类都是类型构造器。那些不带类型参数的，可以看做带了零个类型参数的“参数化类型”。

### 14.1.3 类型参数的名称

请使用描述性的词来命名类型参数。Scala 初学者往往抱怨，在 Scala 的实现以及 Scaladoc 中类型参数名称太过简单，如：给 `List.+:` 方法的类型参数命名为 `A` 和 `B`。不过，你很快就能学会如何解释这些符号，它遵循一些简单的规则。

(1) 为非常通用的类型参数（例如：表示容器元素的类型参数），使用 `A`、`B`、`T1`、`T2` 等单字母或双字母的名字。请注意，元素的类型与容器的类型并没有太紧密的联系。无论 `List` 保存的是字符串、数字或其他字符串，都不影响其工作方式。这一解耦（decouple）使得“泛型编程”成为可能。

(2) 对那些与容器密切相关的类型使用更具描述性的名称。也许，当你第一次遇到 `List.+:` 这个方法签名时，它并没有表现出明显的意义，但一旦学习了 12.3 节中我们对设计常用方法的讨论，你就会明白 `List.+:` 的含义。

## 14.2 类型边界

当定义一个参数化的类型或方法时，可能需要指定类型参数的*边界*（bound）。例如：容器可能会假定，其类型参数都支持某一种方法。

## 14.2.1　类型边界上限

类型边界上限是指，某一类型必须是另一种类型的子类型。在 5.7 节，Predef 为 Array（Java 数组）定义了隐式的包装类型 collection.mutable.ArrayOps，它提供了我们熟知和喜爱的序列操作。

我们定义了好几个转换，大部分是对特定 AnyVal 类型的包装，如 Long。其中对 Array[AnyRef] 类型的转换是个例外：

```
implicit def refArrayOps[T <: AnyRef](xs: Array[T]): ArrayOps[T] =
  new ArrayOps.ofRef[T](xs)
implicit def longArrayOps(xs: Array[Long]): ArrayOps[Long] =
  new ArrayOps.ofLong(xs)
... // 其他AnyVal类型的方法。
```

类型参数 A <: AnyRef 表示"任何 A 都是 AnyRef 的子类型"。回想一下，一个类型是其本身的子类型和父类型，所以 A 也可以是 AnyRef 本身。所以，<: 操作符表示其左边的类型必须派生自右边的类型，或者两者是同一类型。我们在 2.7 节中谈到的 <: 操作符实际上是 Scala 的保留字。

第一个方法被限制为只对 AnyRef 的子类型适用，第二个方法则只对 Long 这个特定类型起作用，这样，在隐式转换时两者就没有歧义了。

这些边界被称为类型边界上限，继承关系一般依据类型继承结构图而定。在类型继承结构图中，子类位于父类下方。我们遵循 10.2 节中"Scala 类型结构"图所示的基础关系。

类型边界和变异标记涵盖的是两个不相干的问题。类型边界对参数化类型所允许采用的类型做了限制，如 T <: AnyRef 约束 T 必须为 AnyRef 的子类型。变异标记表示参数化类型的子类实例是否可以替换父类实例。例如，由于 List[+T] 是协变的，所以 List[String] 是 List[Any] 的子类型。

## 14.2.2　类型边界下限

与类型边界上限相反，类型边界下限表示某个类型必须是另一个类型的父类（或该类型本身）。Option 中的 getOrElse 方法就是一个例子：

```
sealed abstract class Option[+A] extends Product with Serializable {
  ...
  @inline final def getOrElse[B >: A](default: => B): B = {...}
  ...
}
```

如果 Option 实例是 Some[A]，方法就返回 Some[A] 包含的值。否则，就对命名参数 default 求值，并将其返回。B 应该是 A 的父类型，为什么呢？为什么 Scala 要求这个方法必须声明成这种格式呢？我们考虑下面这个例子，尝试理解原因：

```
// src/main/scala/progscala2/typesystem/bounds/lower-bounds.sc

class Parent(val value: Int) {                     // ❶
```

```
  override def toString = s"${this.getClass.getName}($value)"
}
class Child(value: Int) extends Parent(value)

val op1: Option[Parent] = Option(new Child(1))   // ❷     Some(Child(1))
val p1: Parent = op1.getOrElse(new Parent(10))   // 结果: Child(1)

val op2: Option[Parent] = Option[Parent](null)   // ❸     None
val p2a: Parent = op2.getOrElse(new Parent(10))  //结果: Parent(10)
val p2b: Parent = op2.getOrElse(new Child(100))  //结果: Child(100)

val op3: Option[Parent] = Option[Child](null)    // ❹     None
val p3a: Parent = op3.getOrElse(new Parent(20))  //结果: Parent(20)
val p3b: Parent = op3.getOrElse(new Child(200))  //结果: Child(200)
```

❶ 为了举例，定义了一个简单的继承树。

❷ op1 是个引用，它只知道自己指向 Option[Parent]，并不知道指向的对象其实是 Option[Parent] 的子类型 Option[Child]。由于 Option[+T] 是协变的，因此 Option[Child] 是 Option[Parent] 的子类型。

❸ Option[X](null) 总是返回 None。在这里，返回的引用是 Option[Parent] 类型的。

❹ 又是返回 None，尽管赋值给了一个 Option[Parent] 类型的引用，但引用的类型是 Option[Child]。

最后有两行代码很关键：

```
val op3: Option[Parent] = Option[Child](null)
val p3a: Parent = op3.getOrElse(new Parent(20))
```

op3 这一行显式地说明了 Option[Child](null)（即 None）被赋值给了 Option[Parent]。但如果赋值来自于一个“黑盒”的方法调用呢？那样的话我们并不知道其真实类型到底是什么。这个例子的关键在于，客户端调用只持有对 Option[Parent] 的引用，所以客户端调用有合理的理由认为它可以从 Option[Parent] 中提取一个 Parent 值，无论实际值是 None 还是 Some[Parent]。如果是 None，则返回默认的 Parent 参数；如果是 Some[Parent]，则返回的是 Some 中的值。所有情况都会返回一个 Parent 类型的值，正如我们所看到的那样，尽管有时返回的实际上是 Child 子类的实例。

假设 getOrElse 的声明是这样：

```
@inline final def getOrElse(default: => A): A = {...}
```

这样的话，调用 op3.getOrElse(new  Parent(20)) 将无法通过类型检查。因为 op3 指向的对象是 Option[Child]，所以传给 getOrElse 的期望类型是 Child 的实例。

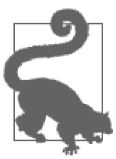
这就是编译器不允许上述这种简单的方法签名，而需要采用 [B >: A] 边界标记的签名的原因。我们自定义一个类似 Option 的类型，名为 Opt，并使用类似上述的方法签名。为简单起见，我们用 null 值代替 None，它是 getOrElse 方法的默认返回值：

```
// src/main/scala/progscala2/typesystem/bounds/lower-bounds2.sc
scala> case class Opt[+A](value: A = null) {
     |  def getOrElse(default: A) = if (value != null) value else default
```

```
     |}
<console>:8: error: covariant type A occurs in contravariant position
  in type A of value default
         def getOrElse(default: A) = if (value != null) value else default
                       ^
```

如上述示例所示，每当你看到 covariant type A occurs in contravariant position 的错误信息时，这说明你曾尝试定义一个协变的参数化类型，但你同时想要定义一个接受该类型作为参数的方法，而不是接受一个该类型的父类作为参数，即 `B >: A`。根据前文所述，这种做法是不允许的。

这种类型的参数看起来似乎很熟悉，的确如此。它实际上是我们在 10.1.1 节中讨论过的函数参数类型。这种类型必须是逆变的，如 `Function2[-A1, -A2, +R]`，因为这些参数类型出现在 `apply` 方法的逆变位置。



试图理解变异标记和类型边界的工作方式，我们应当从调用方的角度了解这些类型的实例发生了什么。调用时，引用可能指向父类，但该实例其实是个子类实例。

考虑一下，如果我们将协变的 `Opt[+A]` 改为非变异的 `Opt[A]`，会发生什么呢？

```
// src/main/scala/progscala2/typesystem/bounds/lower-bounds2.sc
scala> case class Opt[A](value: A = null) {
     |  def getOrElse(default: A) = if (value != null) value else default
     |}

scala> val p4: Parent = Opt(new Child(1)).getOrElse(new Parent(10))
<console>:11: error: type mismatch;
 found   : Parent
 required: Child
       val p4: Parent = Opt(new Child(1)).getOrElse(new Parent(10))
                                                    ^

scala> val p5: Parent = Opt[Parent](null).getOrElse(new Parent(10))
p5: Parent = Parent(10)

scala> val p6: Parent = Opt[Child](null).getOrElse(new Parent(10))
<console>:11: error: type mismatch;
 found   : Parent
 required: Child
       val p6: Parent = Opt[Child](null).getOrElse(new Parent(10))
                                                   ^
```

代码只对 `pa5` 的赋值正常工作，我们无法将 `Opt[Child]` 类型的实例赋值给 `Opt[Parent]` 的引用。

还有一种例子的奥秘值得讨论。在有些方法中，当我们向一个不可变集合添加新元素以构造一个新的集合时，其类型参数必须具有逆变的行为，但传入的是协变的参数化类型。

`Seq.+:` 方法用来给序列在头部插入新元素，返回插入后生成的新序列。我们之前已经使用过这个方法，一般都通过 `+:` 操作符来调用，如：

```
scala> 1 +: Seq(2, 3)
res0: Seq[Int] = List(1, 2, 3)
```

Scaladoc 中给出的是简化的方法签名，它假定我们插入的元素与其他元素类型相同，均为 A。但方法的实际声明更通用些，这里给出了这两种声明：

```
def +:(elem: A): Seq[A] = {...}                          // ❶
def +:[B >: A, That](elem: B)(                           // ❷
  implicit bf: CanBuildFrom[Seq[A], B, That]): That = {...}
```

❶ 简化的方法签名，假设类型参数 A 不变。

❷ 实际的签名，允许插入元素的类型为 A 的任意父类型，包括之前讨论过的 CanBuildFrom 形式也是允许的。

在下面的例子中，我们向 Seq[Int] 序列插入一个 Double 值：

```
scala> 0.1 +: res0
<console>:9: warning: a type was inferred to be `AnyVal`; this may
  indicate a programming error.
              0.1 +: res0
                  ^
res1: Seq[AnyVal] = List(0.1, 1, 2, 3)
```

如果你用的 Scala 是 2.11 版本之前的，你将不会看到这条警告，稍后我会解释原因。

元素类型 B 与插入的元素类型 Double 不同，B 被推断为最近类型上限（least upper bound，LUB），也就是原始元素类型 A（Int）与新元素类型 Double 的最近公共父类。所以，B 被推断为 AnyVal 类型。

对于 Option，B 被推断为与默认参数相同的类型。如果传入的对象为 None，返回默认的参数，我们可能就“忘记”了原始类型 A。

对于上面例子中的序列，我们保留了原始的类型 A 的值，然后添加了类型 B 的新值，于是系统推断出 LUB 类型是 A 和 B 的共同父类。

尽管这种隐式推断很方便，但得到一个更宽泛的 LUB 类型可能会使我们吃惊，因为你不想改变原始的类型。这就是为什么 Scala 2.11 在这种情况下添加警告信息的原因。

解决方法是显式地声明期望的返回类型：

```
// Scala 2.11 解决该警告的方法：
scala> val l2: List[AnyVal] = 0.1 +: res0
l2: List[AnyVal] = List(0.1, 1, 2, 3)
```

这样，编译器就知道你希望得到一个更宽泛的 LUB 类型，于是警告就消失了。

总的来说，那些类型参数为协变的参数化类型，与方法参数的类型边界下限关系密切。

最后，你可以同时使用类型边界的上限和下限：

```
class Upper
class Middle1 extends Upper
class Middle2 extends Middle1
class Lower extends Middle2
```

```
case class C[A >: Lower <: Upper](a: A)
// case class C2[A <: Upper >: Lower](a: A)  // 无法通过编译
```

类型参数 A 必须首先出现。注意到 C2 无法通过编译，类型边界的下限必须在上限之前出现。

## 14.3 上下文边界

在 5.1 节，我们提到过上下文边界。我们当时用到的例子如下：

```
// src/main/scala/progscala2/implicits/implicitly-args.sc
import math.Ordering

case class MyList[A](list: List[A]) {
  def sortBy1[B](f: A => B)(implicit ord: Ordering[B]): List[A] =
    list.sortBy(f)(ord)

  def sortBy2[B : Ordering](f: A => B): List[A] =
    list.sortBy(f)(implicitly[Ordering[B]])
}

val list = MyList(List(1,3,5,2,4))

list sortBy1 (i => -i)
list sortBy2 (i => -i)
```

比较 sortBy 的两个版本，隐式参数在 sortBy1 内显式地被给出，而在 sortBy2 内却被隐藏，该隐式参数实际上是一个参数化类型。类型表达式 B : Ordering 与 B 和 Ordering[B] 类的隐式参数是等价的。这意味着，除非存在一个对应的 Ordering[B] 类型，否则 B 就不能作为参数。

与之类似的一个概念是*视图边界*（view bound）。

## 14.4 视图边界

视图边界类似于上下文边界，可以被认为是上下文边界的一个特例。它们可以通过以下任一方式声明：

```
class C[A] {
  def m1[B](...)(implicit view: A => B): ReturnType = {...}
  def m2[A <% B](...): ReturnType = {...}
}
```

与以前的上下文边界的例子对比，在该例中，隐式的 A : B 值必须是 B[A] 类型。在本例中，我们需要一个隐式的函数，将 A 的实例转为 B 的实例。如果可以转换，我们就说"B 是 A 的一个视图"。类型边界上限的表达式 A <: B 表示 A 是 B 的子类，但在这里我们的要求放宽很多，只需要 A 可以转化为 B 即可。

以下是使用这种特性的一个代码示例。Hadoop（http://hadoop.apache.org）的 Java API 要求数据必须可以被包装在自定义的序列化类型里，它实现了一个称为 Writable（https://

hadoop.apache.org/docs/current/api/org/apache/hadoop/io/Writable.html）的接口，用于将数据发送到远程进程。这个 API 的用户必须显式地使用 `Writable` 接口，虽然这有些不太方便。我们可以用视图边界来自动地处理（简单起见，我们用自己的 `Writable` 代替 Hadoop 的 `Writable`）：

```
// src/main/scala/progscala2/typesystem/bounds/view-bounds.sc
import scala.language.implicitConversions

object Serialization {
  case class Writable(value: Any) {
    def serialized: String = s"-- $value --"                          // ❶
  }

  implicit def fromInt(i: Int) = Writable(i)                           // ❷
  implicit def fromFloat(f: Float) = Writable(f)
  implicit def fromString(s: String) = Writable(s)
}

import Serialization._

object RemoteConnection {                                               // ❸
  def write[T <% Writable](t: T): Unit =                                // ❹
    println(t.serialized)  // Use stdout as the "remote connection"
}

RemoteConnection.write(100)       // Prints -- 100 --
RemoteConnection.write(3.14f)     // Prints -- 3.14 --
RemoteConnection.write("hello!")  // Prints -- hello! --
// RemoteConnection.write((1, 2))                                         ❺
```

❶ 简单起见，用 `String` 作为“二进制”数据的格式。

❷ 定义几个隐式转换规则。需要注意，这里定义的是方法，但我们需要类型为 `A => B` 的函数。Scala 会帮助我们在需要的时候将方法提升为函数。

❸ 该对象封装了向“远程”连接写的数据。

❹ 定义一个方法，接受任意类型的实例，并将其写入到远程连接。该方法会触发隐式转换，引起 `serialized` 方法的调用。

❺ 不能使用元组，因为没有可用的“视图”。

这里我们不需要使用 `Predef.implictly`（http://www.scala-lang.org/api/current/scala/Predef$.html）或其他类似的东西，隐式转换由编译器为我们自动触发。

视图边界可以用上下文边界实现。尽管视图边界提供了更简洁的语法，但上下文边界更通用。因此，在 Scala 社区中出现了一些废弃视图边界的讨论。之前的例子使用上下文边界重新设计后的结果如下：

```
// src/main/scala/progscala2/typesystem/bounds/view-to-context-bounds.sc
import scala.language.implicitConversions

object Serialization {
  case class Rem[A](value: A) {
```

```
    def serialized: String = s"-- $value --"
  }
  type Writable[A] = A => Rem[A]                                // ❶
  implicit val fromInt: Writable[Int]       = (i: Int)    => Rem(i)
  implicit val fromFloat: Writable[Float]   = (f: Float)  => Rem(f)
  implicit val fromString: Writable[String] = (s: String) => Rem(s)
}

import Serialization._

object RemoteConnection {
  def write[T : Writable](t: T): Unit =                         // ❷
    println(t.serialized)  // Use stdout as the "remote connection"
}

RemoteConnection.write(100)       // 打印 -- 100 --                ❸
RemoteConnection.write(3.14f)     // 打印 -- 3.14 --
RemoteConnection.write("hello!")  // 打印 -- hello! --
// RemoteConnection.write((1, 2))
```

❶ A 型的别名使得上下文边界更容易使用。后面几行与前面的例子相同，是隐式转换的定义。

❷ 用上下文边界实现的 `write` 方法。

❸ 与前面的例子相同，调用 `write`，产生相同的输出结果。

所以，尽量避免使用视图边界，因为它们可能在未来的版本中被废弃掉。

# 14.5 理解抽象类型

参数化类型在静态的、面向对象的语言中很常见。除了参数化类型，Scala 还支持抽象类型，这在函数式语言中很常见。我们在 2.13 节介绍过抽象类型。参数化类型和抽像类型存在一定的重叠，稍后我们会探讨这一点。我们先来讨论抽象类型的使用：

```
// src/main/scala/progscala2/typesystem/abstracttypes/abstract-types-ex.sc

trait exampleTrait {
  type t1               // t1没有任何约束
  type t2 >: t3 <: t1   // t2必须是t3的父类,t1的子类
  type t3 <: t1         // t3必须是t1的子类
  type t4 <: Seq[t1]    // t4 必须是t1的序列的子类
  // type t5 = +AnyRef  // 错误:不能使用变异标记

  val v1: t1            // t1定义后才能初始化
  val v2: t2            // 同上…
  val v3: t3            // ...
  val v4: t4            // ...
}
```

注释已经说明了大部分的细节。`t1`、`t2` 和 `t3` 之间的关系很有趣。首先，`t2` 的声明表明它必须处于 `t1` 和 `t3` 的“中间”。`t1` 无论是什么类型，都必须是 `t2` 的超类（或等于 `t2`），而 `t3` 必须被设定成 `t2` 的子类（或等于 `t2`）。

注意用于声明 t3 的那一行代码。为了与 t2 的声明保持一致，t3 必须声明为 t1 的子类型。如果省略类型边界的标记，错误会被引发。这是因为之前定义的 t2 已经可以推断出 t3<: t1。使用 t3 <: t2 会在 t2 >: t3 <: t1 这行触发一个错误信息：“对 t2 非法循环引用（illegal cyclic reference to t2）”。我们不能省略对 t3 的声明，也不能认为 t3 可以从 t2 的声明“推断”得到。当然，这种复杂的例子是由于演示的需要，人为构造出来的。

我们不能在类型成员上使用变异标记。注意到，这里的类型是封装类型中的类型成员，不是参数化类型中的类型参数。封装的类型可能与其他类型存在继承关系，但其类型成员的行为就像成员方法和成员变量一样，它们不影响其所在类型的继承关系。同其他成员一样，成员类型既可以被声明为抽象的，也可以被声明为具体的。然而，与成员变量和成员方法不同，在子类中成员类型可以被重新定义，即使定义并不完全。当然，只有在所有抽象类型都给出具体定义之后，才能创建类型的实例。

我们定义一些 trait 和一个类来测试一下这些类型：

```
trait T1 { val name1: String }
trait T2 extends T1 { val name2: String }
case class C(name1: String, name2: String) extends T2
```

最后，我们可以定义一个具体类型，定义抽象类型成员，并初始化相应的值：

```
object example extends exampleTrait {
  type t1 = T1
  type t2 = T2
  type t3 = C
  type t4 = Vector[T1]

  val v1 = new T1 { val name1 = "T1"}
  val v2 = new T2 { val name1 = "T1"; val name2 = "T2" }
  val v3 = C("1", "2")
  val v4 = Vector(C("3", "4"))
}
```

## 比较抽象类型与参数化类型

从技术上讲，你几乎可以使所有的参数化类型支持抽象类型，反之亦然。然而，在实践中，不同的特征适合处理不同的设计问题。

参数化类型可以很好地用于容器中，如集合。而类型参数所表示的元素类型与容器本身之间并没有什么联系。例如，字符串列表、浮点数列表与整数列表的工作方式相同。

如果换成用抽象类型会如何？以下 Some 的声明是从标准库中摘录的：

```
case final class Some[+A](val value : A) { ... }
```

如果我们试图将其换成抽象类型，将会变成：

```
case final class Some(val value : ???) {
  type A
  ...
}
```

参数 value 应该用什么类型呢？我们不能使用 A，因为它不在构造器参数列表的作用域内。我们虽然可以用 Any，但这样就违背了类型安全的目的。

所以，当类型的参数用于构造器时，唯一合适的选择就是参数化类型。

反过来，抽象类型在相互联系密切的“类型家族”中也非常有用。回想一下我们在 2.13 节见过的例子（这里省略了一些不重要的细节）：

```
// src/main/scala/progscala2/typelessdomore/abstract-types.sc

import java.io._

abstract class BulkReader {
  type In
  val source: In
  def read: String  // Read and return a String
}

class StringBulkReader(val source: String) extends BulkReader {
  type In = String
  def read: String = source
}

class FileBulkReader(val source: File) extends BulkReader {
  type In = File
  def read: String = {...}
}
```

BulkReader 声明了不带类型边界的抽象类型 In。子类 StringBulkReader 和 FileBulkReader 定义了该类型。注意，用户不再通过参数化类型来指定类型。相反，我们对类型成员 In 和包含 In 的类具有完全的控制，所以实现中可以保持一致。

下面我们考虑另一个例子，该示例是*观察者模式*（observer pattern）的一个潜在设计方法。我们曾在 9.2 和 11.4 两节中遇到过观察者模式。第一种方法将会失败，不过我们会在下一节自类型标记中对其进行修复：

```
// src/main/scala/progscala2/typesystem/abstracttypes/SubjectObserver.scalaX
package progscala2.typesystem.abstracttypes

abstract class SubjectObserver {                                    // ❶
  type S <: Subject                                                 // ❷
  type O <: Observer

  trait Subject {                                                   // ❸
    private var observers = List[O]()

    def addObserver(observer: O) = observers ::= observer

    def notifyObservers() = observers.foreach(_.receiveUpdate(this))// ❹
  }

  trait Observer {                                                  // ❺
    def receiveUpdate(subject: S)
```

```
  }
}
```

❶ 将主题 – 观察者关系封装在一个类型里。

❷ 声明主题和观察者的抽象类型成员，该类型成员受接下来声明的 Subject 和 Observer 特征的限制。

❸ Subject 特征，其中维护了观察者列表。

❹ 通知观察者，这一行不能通过编译。

❺ Observer 特征，有一个用于接收更新的方法。

试图编译该文件会产生以下错误信息：

```
em/abstracttypes/observer.scala:14: type mismatch;
[error]  found   : Subject.this.type (with underlying type
                   SubjectObserver.this.Subject)
[error]  required: SubjectObserver.this.S
[error]     def notifyObservers = observers foreach (_.receiveUpdate(this))
[error]                                                               ^
```

我们想要主题和观察者都使用有边界的的抽象类型成员，这样，当我们为抽象类型成员指定具体类型，特别是 S 类型时，Observer.receiveUpdate(subject：S) 会得到准确的 S 类型，而不是没什么用处的父类型 Subject。

然而，当我们编译时，传递给 receiveUpdate 的 this 是 Subject 类，而不是更具体的类型 S。

不过，我们可以用自类型标记来修复这个问题。

## 14.6 自类型标记

我们可以在方法中用 this 来指代调用该方法的实例，这对于引用实例的其他成员来说很有用。通常，我们不需要显式地使用 this，但如果作用域内有多个名称相同的变量时，显式地使用 this 有助于消除二义性。

自类型标记（self-type annotation）可以达到两个目标。首先，它允许为 this 指定额外的类型期望。其次，它可以被用于创建 this 的别名。

为了说明如何添加额外的类型期望，我们重新回顾下之前的 SubjectObserver。通过指定类型期望，我们将能够解决遇到的编译问题。但是需要改动两处内容：

```
// src/main/scala/progscala2/typesystem/selftype/SubjectObserver.scala
package progscala2.typesystem.selftype

abstract class SubjectObserver {
  type S <: Subject
  type O <: Observer

  trait Subject {
    self: S =>                                              // ❶
```

```
    private var observers = List[O]()

    def addObserver(observer: O) = observers ::= observer

    def notifyObservers() = observers.foreach(_.receiveUpdate(self))// ❷
  }

  trait Observer {
    def receiveUpdate(subject: S)
  }
}
```

❶ 为 Subject 声明一个自类型标记 self: S。这意味着我们现在可以“假设”Subject 为子类型 S 的实例。S 实际上是混入了 Subject 特征的任何类型。

❷ 给 receiveUpdate 传入 self 而不是 this。

现在，代码可以通过编译。下面我们来观察类型是如何统计按钮点击次数的：

```
// src/main/scala/progscala2/typesystem/selftype/ButtonSubjectObserver.scala
package progscala2.typesystem.selftype

case class Button(label: String) {                                  // ❶
  def click(): Unit = {}
}

object ButtonSubjectObserver extends SubjectObserver {              // ❷
  type S = ObservableButton
  type O = ButtonObserver

  class ObservableButton(label: String) extends Button(label) with Subject {
    override def click() = {
      super.click()
      notifyObservers()
    }
  }

  trait ButtonObserver extends Observer {
    def receiveUpdate(button: ObservableButton)
  }
}

import ButtonSubjectObserver._

class ButtonClickObserver extends ButtonObserver {                  // ❸
 val clicks = new scala.collection.mutable.HashMap[String,Int]()

  def receiveUpdate(button: ObservableButton) = {
    val count = clicks.getOrElse(button.label, 0) + 1
    clicks.update(button.label, count)
  }
}
```

❶ 简单的 Button 类。

❷ SubjectObserver 的一个具体子类，用来表示按钮。其中 Subject 和 Observer 都被细化为我们想要的更具体的子类（注意传递给 ButtonObserver.receiveUpdate 值的类型）。传递给 ButtonObserver.receiveUpdate 值的类型覆写了 Button.click，用来调用 Button.click 后通知观察者。

❸ 实现 ButtonObserver，用来跟踪 UI 中每个按钮的点击次数。

下面脚本创建的两个 ObservableButton 实例，与同一个观察者绑定，对这两个按钮进行若干次点击，然后打印出每个按钮的点击次数：

```
// src/main/scala/progscala2/typesystem/selftype/ButtonSubjectObserver.sc
import progscala2.typesystem.selftype._

val buttons = Vector(new ObservableButton("one"), new ObservableButton("two")
val observer = new ButtonClickObserver
buttons foreach (_.addObserver(observer))
for (i <- 0 to 2) buttons(0).click()
for (i <- 0 to 4) buttons(1).click()
println(observer.clicks)
// Map("one" -> 3, "two" -> 5)
```

因此，我们可以使用自类型标记来解决使用抽象类型成员时带来的问题。

另一个例子是用于指定“模块”并将“模块”连接在一起的模式。以下的例子给出了一个三层的应用程序，分别是持久层、中间层和 UI 层：

```
// src/main/scala/progscala2/typesystem/selftype/selftype-cake-pattern.sc

trait Persistence { def startPersistence(): Unit }                  // ❶
trait Midtier { def startMidtier(): Unit }
trait UI { def startUI(): Unit }

trait Database extends Persistence {                                // ❷
  def startPersistence(): Unit = println("Starting Database")
}
trait BizLogic extends Midtier {
  def startMidtier(): Unit = println("Starting BizLogic")
}
trait WebUI extends UI {
  def startUI(): Unit = println("Starting WebUI")
}

trait App { self: Persistence with Midtier with UI =>               // ❸

  def run() = {
    startPersistence()
    startMidtier()
    startUI()
  }
}

object MyApp extends App with Database with BizLogic with WebUI     // ❹

MyApp.run                                                           // ❺
```

❶ 为应用程序的持久层、中间层、UI 层定义 trait。

❷ 用具体的 trait 实现三层的行为 。

❸ 定义一个 trait（或者使用抽象类型），将各层连接在一起。在这个简化的例子中，`run` 方法只用于将各层启动。而提到的自类型标记将在下文中进行讨论。

❹ 定义的 `MyApp` 对象继承了 `App` 特征，并混入了实现三层逻辑的具体 trait。

❺ 运行应用程序。

运行此脚本将打印以下输出：

```
Starting Database
Starting BizLogic
Starting WebUI
```

该脚本展示了一个支持多层次的应用程序的基础设置。每个抽象 trait 都声明了 `start*` 方法，用来完成各个层次的初始化。每个抽象层的逻辑都对应一个具体的 trait，而不是使用类来实现，因此我们可以像混入类型一样使用这些层次。

`App` 特征将各个层次连接在一起，并通过 `run` 方法启动各个层次。需要注意的是，这里没有为这些 trait 指定具体的实现。一个具体的应用程序必须通过混入这些 trait 的实现来进行构造。

自类型标记是这里的关键：

```
self: Persistence with Midtier with UI =>
```

为自类型标记添加的类型标记，如：本例的 `Persistence with Midtier with UI`，指定了该 trait 或抽象类在定义具体实例时，必须混入这些 trait 或实现抽象类型成员的子类。因为有这个假设，trait 被允许访问所混入的特征成员（尽管此时它们还不是本类型的成员）。在这里，`App.run` 调用来自其他特征的 `start*` 方法。

具体的实例 `MyApp` 继承了 `App`，并混入了满足依赖条件的 trait。

这一层次的栈堆叠模式被称为*蛋糕模式*（cake pattern），其中的模块用 trait 来声明，另一个抽象类型则用于将这些 trait 和自类型标记整合在一起。具体对象混入了 trait 的具体实现，继承了可选的父类（我们将在 23.3 节中详细地讨论这个模式，阐述对该模式的赞成和反对观点）。

使用自类型标记实际上与使用继承和混入等价（除了没有定义 `self` 以外）：

```
trait App extends Persistence with Midtier with UI {
  def run = { ... }
}
```

也有一些特殊情况下，自类型标记的行为不同于继承。但在实践中，这两种方法可以相互替换使用。

事实上，这两种方法表达了不同的意图。刚刚展示的基于继承的实现表明应用程序是 `Persistence`、`Midtier` 和 `UI` 的一个子类型。与此相反，自类型标记则更加明确地表示其行为的组合是通过混入实现的。

自类型标记强调用混入实现组合。继承意味着类之间是父类与子类的关系。

这就是说，除非需要继承大规模的“模块”（trait），且自类型标记能更清楚地表明设计思路的情况，否则大部分 Scala 代码倾向于使用继承的方法，而不是自类型标记。

现在我们来考虑自类型标记的第二种用途，即：对 this 做别名：

```
// src/main/scala/progscala2/typesystem/selftype/this-alias.sc

class C1 { self =>                                                // ❶
  def talk(message: String) = println("C1.talk: " + message)
  class C2 {
    class C3 {
      def talk(message: String) = self.talk("C3.talk: " + message)  // ❷
    }
    val c3 = new C3
  }
  val c2 = new C2
}
val c1 = new C1
c1.talk("Hello")                                                  // ❸
c1.c2.c3.talk("World")                                            // ❹
```

❶ 在 C1 所处的上下文中将 self 定义为 this 的别名。

❷ 用 self 调用 C1.talk。

❸ 用 c1 实例调用 C1.talk。

❹ 用 c1.c2.c3 实例调用 C3.talk，C3.talk 调用 C1.talk。

注意，在这里 self 这个名字是可以任意取的，并不是关键字。你可以用任何合法的名字。如果需要的话，我们也可以在 C2 和 C3 里定义自类型标记。

脚本打印的输出如下：

```
C1.talk: Hello
C1.talk: C3.talk: World
```

如果没有自类型标记，我们就不能直接从 C3.talk 中调用 C1.talk，因为两者名称相同，后者屏蔽了前者。C3 也不是 C1 的直接子类，所以也不能调用 super.talk。

所以，在这里，你可以认为自类型标记是一个“通用的 this”引用。

## 14.7　结构化类型

你可以将结构化类型（structual type）当作一种类型安全的鸭子类型（duck typing）。鸭子类型是动态类型语言中的一种方法解析方式。（“如果它走起来像鸭子，说话像鸭子，那么它一定是鸭子”。）例如：在 Ruby 中，当你的代码包含 starFighter.shootWeapons 时，代码在运行的时候其实并不知道 starFighter 实例是否存在于 shootWeapons 方法中，但它会

遵循各种规则来寻找方法以供调用，或者在找不到方法时对失败进行处理。

Scala 不支持这种运行时的方法解析（在第 19 章中会讨论这条规则的一个例外）。相反，Scala 支持编译时的一个类似的机制。Scala 允许你指定对象必须符合某种特定的结构：必须包含特定成员（类型、字段或方法），但不要求指定封装了这些成员的类型具有什么名称。

我们通常称之为*指名类型*（nominal typing），因为对应的类型具有名称。在结构化类型中，我们只考虑该类型的结构。所以它可以是匿名的。

下面我们通过一个例子来查看如何在观察者模式中使用结构化类型。我们先从 9.2 节中的简单实现入手，而不是本章前文中使用的实现。以下是该示例中的重点细节：

```
trait Observer[-State] {
  def receiveUpdate(state: State): Unit
}
trait Subject[State] {
  private var observers: List[Observer[State]] = Nil
  ...
}
```

这个实现的一个缺点是，任何想要观察 `Subject` 状态变化的类型都必须实现 `Observer` 特征。但实际上，真正的最低要求是实现 `receiveUpdate` 方法。

下面的代码通过在 `Observer` 中使用结构化类型，重新实现了这个例子：

```
// src/main/scala/progscala2/typesystem/structuraltypes/Observer.scala
package progscala2.typesystem.structuraltypes

trait Subject {                                                  // ❶

  import scala.language.reflectiveCalls                          // ❷

  type State                                                     // ❸

  type Observer = { def receiveUpdate(state: Any): Unit }        // ❹

  private var observers: List[Observer] = Nil                    // ❺

  def addObserver(observer:Observer): Unit =
    observers ::= observer

  def notifyObservers(state: State): Unit =
    observers foreach (_.receiveUpdate(state))
}
```

❶ 与主题不相关的修改，不过具有说明作用。去掉了之前的类型参数 `State`，改而使用抽象类型。

❷ 启用可选的反射方法调用特性（见下面的说明）。

❸ `State` 抽象类型。

❹ `Observer` 是一个结构化类型。

❺ State 类型参数也已从 Observer 中移除。

声明 type Observer = { def receiveUpdate(subject: Any): Unit } 表示任何有这种 receiveUpdate 方法的对象都可以作为一个观察者。不幸的是，Scala 不会让结构化类型指向抽象类型或类型参数。因此，我们不能使用 State，而必须使用已经知道的类型（如 Any）。这就意味着，接收器可能需要将实例转换为正确的类型（这是一个很大的缺点）。

import 语句暗示了另外一个缺点。因为我们没有类型名可用于验证候选的观察者实例是否实现了正确的方法，编译器必须使用反射来确认该方法存在于该实例中。这会增加运行开销，不过除非经常添加观察者，否则开销并不明显。反射是一项可选功能，因此我们要使用 import 语句导入才能支持该功能。

以下代码尝试了一种新的实现：

```
// src/main/scala/progscala2/typesystem/structuraltypes/Observer.sc
import progscala2.typesystem.structuraltypes.Subject
import scala.language.reflectiveCalls

object observer {                                              // ❶
  def receiveUpdate(state: Any): Unit = println("got one! "+state)
}

val subject = new Subject {                                    // ❷
  type State = Int
  protected var count = 0

  def increment(): Unit = {
    count += 1
    notifyObservers(count)
  }
}

subject.increment()
subject.increment()
subject.addObserver(observer)
subject.increment()
subject.increment()
```

❶ 用正确的方法声明观察者对象。

❷ 实例化 State 特征，提供 State 抽象类型的定义和其他行为。

需要注意的是，两次 increment 发生之后，我们才注册观察者，所以观察者只会打印它接收到的数字 3 和 4。

尽管存在缺点，结构化类型有降低耦合的特性。在这个例子中，存在的耦合只包括一个方法签名，而不是一个类型，如一个共享的 trait。

最后再看一下这个例子，我们还是需要耦合到一个特定的名字上，即方法 receiveUpdate！在某种意义上说，我们只是将耦合从类型名称转移到了方法名称。这个名称完全是任意的，所以我们可以在解耦上更进一步，定义 Observer 为某个单参数函数的别名。下面是本示例的最终形式：

```
// src/main/scala/progscala2/typesystem/structuraltypes/SubjectFunc.scala
package progscala2.typesystem.structuraltypes

trait SubjectFunc {                                                  // ❶

  type State

  type Observer = State => Unit                                      // ❷

  private var observers: List[Observer] = Nil

  def addObserver(observer:Observer): Unit =
    observers ::= observer

  def notifyObservers(state: State): Unit =                          // ❸
    observers foreach (o => o(state))
}
```

❶ 给 Subject 定一个新名。给整个文件重命名，因为观察者已经不那么重要了！

❷ 定义 Observer 为函数 State => Unit 的别名。

❸ 通知各个观察者，意味着调用它们的 apply 方法。

测试代码几乎相同，以下只列出不同的地方：

```
// src/main/scala/progscala2/typesystem/structuraltypes/SubjectFunc.sc

import progscala2.typeSystem.structuraltypes.SubjectFunc

val observer: Int => Unit = (state: Int) => println("got one! "+state)

val subject = new SubjectFunc { ... }
```

这样就好多了！所有基于名称的耦合都不见了，这样我们就不再需要反射调用，同时函数的参数类型也能再次使用 State 而不是 Any 了。

但是这并不意味着结构化类型没有用。我们的示例只需要一个函数来实现我们所需要的功能。在一般情况下，一个结构类型可能有好几个成员，一个匿名函数可能不足以满足我们的需要。

## 14.8 复合类型

当你声明一个组合了若干种类型的实例时，你就得到了*复合类型*（compound type）：

```
trait T1
trait T2
class C
val c = new C with T1 with T2  // c的类型：C with T1 with T2
```

在这里，c 的类型是 C with T1 with T2，这是另一种声明类型的方法，该类型继承了 C，并混入了 T1 和 T2。所以，c 被认为是所有三种类型的子类型：

```
val t1: T1 = c
val t2: T2 = c
val c2: C  = c
```

## 类型细化

类型细化（type refinement）是复合类型的附加部分。它们都与从 Java 中获知的一种思想有关，即通过提供匿名内部类实现某些接口，以添加方法实现和可选的其他成员。

如果你有一个由 C 类对象组成的 java.util.List（http://docs.oracle.com/javase/8/docs/api/java/util/List.html）列表，那么对于某些类，你可以使用静态方法 java.util.Collections.sort（http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html）来为列表就地排序：

```
List<C> listOfC = ...
java.util.Collections.sort(listOfC, new Comparator<C>() {
  public int compare(C c1, C c2) {...}
  public boolean equals(Object obj) {...}
});
```

我们“细化”类型 Comparator 用于创建一个新的类型。JVM 会在字节码中为它合成一个唯一的名字。

相比之下，Scala 更进一步，它会合成新类型，并利用新的类型反映我们添加的东西。例如：回顾上一节的结构化类型，我们注意 REPL 返回的类型（对输出做了排版）：

```
scala> val subject = new Subject {
     |   type State = Int
     |   protected var count = 0
     |   def increment(): Unit = {
     |     count += 1
     |     notifyObservers(count)
     |   }
     | }
subject: TypeSystem.structuraltypes.Subject{
  type State = Int; def increment(): Unit} = $anon$1@4e3d11db
```

类型的签名中添加了额外的结构化组件。

类似地，当实例化对象同时混入 trait 时，我们就创建了一个细化的类型。考虑以下示例，我们在类型中混入了 logging 特征（省略了部分细节）：

```
scala> trait Logging {
     |   def log(message: String): Unit = println(s"Log: $message")
     | }

scala> val subject = new Subject with Logging {...}
subject: TypeSystem.structuraltypes.Subject with Logging{
  type State = Int; def increment(): Unit} = $anon$1@8b5d08e
```

为了从外部访问细化后的特性或成员，你需要使用反射 API（见 24.2 节）。

## 14.9 存在类型

*存在类型*（existential type）是对类型做抽象的一种方法。它可以让你在不知道具体类型的情况下就断言该类型“存在”。通常你不知道该类型是什么，或着当前上下文中你并不需要知道这个类型。

在以下三种场景中，存在类型对 Scala 与 Java 的类型兼容来说尤为重要。

- 泛型的类型参数被 JVM 字节码“抹去”了（称为*类型擦除*）。例如：创建 `List[Int]` 时，我们在字节码中看不到 `Int` 这个类型。所以，在运行时无法根据已知的类型信息区分 `List[Int]` 和 `List[String]`。
- 你可能接触过“裸”类型，如在 Java 5 之前，库中的集合类型没有类型参数（所有的参数都是 `Object` 类型）。
- 当 Java 在泛型中使用通配符来表示多样行为时，实际的类型是未知的。

考虑对 `List[A]` 中对象做匹配的情况。你可能希望定义两个版本的 `double` 函数，其中一个版本输入 `List[Int]`，并返回一个新的 `List[Int]`，`List[Int]` 的元素加倍（乘以 2）；另一个版本接受一个 `List[String]`，通过对整数调用 `toInt`，将字符串（假设这里的字符串代表整数）映射为整数，然后调用第一个版本的 `double`：

```
object Doubler {
  def double(seq: Seq[String]): Seq[Int] = double(seq map (_.toInt))
  def double(seq: Seq[Int]): Seq[Int] = seq map (_*2)
}
```

你将会得到一个编译错误，显示这两个方法“在类型擦除后，方法的类型相同”（have the same type after erasure）。下面给出了一个有点繁琐的解决方法对列表元素进行逐个检查：

```
// src/main/scala/progscala2/typesystem/existentials/type-erasure-workaround.sc

object Doubler {
  def double(seq: Seq[_]): Seq[Int] = seq match {
    case Nil => Nil
    case head +: tail => (toInt(head) * 2) +: double(tail)
  }

  private def toInt(x: Any): Int = x match {
    case i: Int => i
    case s: String => s.toInt
    case x => throw new RuntimeException(s"Unexpected list element $x")
  }
}
```

当处于这样的类型上下文中时，表达式 `Seq[_]` 是存在类型 `Seq[T] forSome { type T }` 的简写。这是最通用的例子。也就是说，表示列表的类型参数可以是任意类型。表 14-1 列出了使用类型边界的其他一些例子：

表14-1：存在类型的使用示例

| 简　写 | 完整形式 | 描　述 |
|---|---|---|
| Seq[_] | Seq[T] forSome {type T} | T 可以是 Any 的任意子类 |
| Seq[_ <: A] | Seq[T] forSome {type T <: A} | T 可以是 A（在某处已经定义了）的任意子类 |
| Seq[_ >: Z <: A] | Seq[T] forSome {type T >: Z <: A} | T 可以是 A 的子类且是 Z 的超类 |

如果你有考虑 Scala 的泛型语法与 Java 的语法如何对应，你可能已经注意到类似 `java.util.List[_ <: A]` 的表达式在结构上与 Java 的表达式 `java.util.List<? extends A>` 很像。事实上，它们是同一个声明。尽管我们说 Scala 的变异行为在声明时就已经定义了，但你可以用存在类型定义出调用时才确定的变异行为，只是通常很少这么做。

你在 Scala 代码中会经常看到类似 `Seq[_]` 的代码，其中类型参数不能被更具体地指定。但你不会经常看到完整的 `forSome` 形式的存在类型语法。

存在类型存在的主要目的是为了支持 Java 泛型，同时保持它在 Scala 类型系统中的正确性。大多数情况下，类型推断已经为我们隐藏了细节。

## 14.10　本章回顾与下一章提要

这样，我们就完成了 Scala 类型系统特性之旅，这些特性是你写 Scala 代码和使用标准库时最可能遇到的。我们的首要重点是理解面向对象中继承关系的微妙之处，以及变异和类型边界的重要性。下一章我们将继续探索那些不需要尽快掌握的次要功能。

如果你想快速参考 Scala 类型系统及其相关概念，请参见我同事 Konrad Malawski 的 *Scala's Types of Types*（http://ktoso.github.io/scala-types-of-types/）。

# 第15章

# Scala类型系统（II）

本章将紧接着上一章的内容，继续对类型系统进行讲解。假如是 Scala 初学者，你无需立刻理解本章讲解的内容，但是最终你还是会碰到这些问题。当你正忙于 Scala 项目或使用第三方库时，假如碰到了一个从未见过的类型系统概念，那么你也许能在本章找到答案。（如果想了解比本章还要深入的概念，请查看《Scala 语言规范》（http://www.scala-lang.org/docu/files/ScalaReference.pdf）。）目前，我仍然建议你略读本章。尽管无需深入理解这些概念，但是在本书后续的内容中，你还是会看到一些有关路径相关类型（path-dependent type）的更为复杂的示例。

## 15.1 路径相关类型

与之前出现的 Java 语言一样，Scala 允许使用路径表达式对嵌套类型进行访问。

请思考下面的例子：

```
// src/main/scala/progscala2/typesystem/typepaths/type-path.scalaX
package progscala2.typesystem.typepaths

class Service {                                                // ❶
  class Logger {
    def log(message: String): Unit = println(s"log: $message")  // ❷
  }
  val logger: Logger = new Logger
}

val s1 = new Service
val s2 = new Service { override val logger = s1.logger }    // 错误！ ❸
```

❶ 定义了 Service 类，该类中包含嵌套类 Logger。

❷ 为了简化，log 方法中只调用了 println 方法打印日志。

❸ 出现编译错误！

编译该文件时，最后一行代码将产生下列错误：

```
error: overriding value logger in class Service of type this.Logger;
 value logger has incompatible type
       val s2 = new Service { override val logger = s1.logger }
                                                    ^
```

代码中出现的两个 Logge 类型能被视为相同类型吗？答案是否定的。错误信息表示 Scala 期望得到类型为 this.Logger 的 logger 对象。在 Scala 中，每一个 Service 实例的 logger 属性类型都不相同。换言之，logger 的实际类型是*路径相关的*（path-dependent）。下面我们将讨论这种类型路径。

## 15.1.1 C.this

你可以在 C1 类的类体中使用熟悉的 this 关键字，该关键字将指向当前实例。不过此处使用的 this 其实是 Scala 语言中的 C1.this 的缩写：

```
// src/main/scala/progscala2/typesystem/typepaths/path-expressions.scala

class C1 {
  var x = "1"
  def setX1(x:String): Unit = this.x = x
  def setX2(x:String): Unit = C1.this.x = x
}
```

假如 this 出现在类型体内、方法定义体之外，它指向的是类型本身：

```
trait T1 {
  class C
  val c1: C = new C
  val c2: C = new this.C
}
```

很明显，this.C 中出现的 this 引用了 trait T1。

## 15.1.2 C.super

你可以使用 super 关键字引用某一类型的父类：

```
trait X {
  def setXX(x:String): Unit = {} // 函数未执行任何操作!
}
class C2 extends C1
class C3 extends C2 with X {
  def setX3(x:String): Unit = super.setX1(x)
  def setX4(x:String): Unit = C3.super.setX1(x)
  def setX5(x:String): Unit = C3.super[C2].setX1(x)
  def setX6(x:String): Unit = C3.super[X].setXX(x)
  // def setX7(x:String): Unit = C3.super[C1].setX1(x)    // 错误
```

```
    // def setX8(x:String): Unit = C3.super.super.setX1(x)  // 错误
  }
```

在本示例中，C3.super 的作用等同于 super。你也可以使用 [T] 说明当前使用哪个对象的父类，就像 setX5 方法做的那样，setX5 方法选择了 C2 的父类，而 setX6 则选择了 X 的父类。不过，你无法引用“祖父”类型（如 setX7 方法所示），也无法将 super 串行化（如 setX8 所示）。

如果某一类型具有多个祖先，那么当你在该类型中使用未限定类型的 super 引用时，该引用会绑定到哪个类型呢？ super 线性化算法规则决定了 super 指向的目标。（请参考 11.7 节）。

与 this 关键字一样，你也可以在类型体内、方法之外使用 super 引用父类型：

```
class C4 {
  class C5
}
class C6 extends C4 {
  val c5a: C5 = new C5
  val c5b: C5 = new super.C5
}
```

### 15.1.3 path.x

你可以通过使用点号分隔的路径表达式查找嵌套类型。路径表达式中除最后一个元素之外，其他元素都必须保持固定（stable），即这些元素必须是包、单例对象或具有相同别名的类型声明。路径中最后一个元素可以是不固定的，包括类、trait 或者类型成员。请阅读下列示例：

```
package P1 {
  object O1 {
    object O2 {
      val name = "name"
    }
    class C1 {
      val name = "name"
    }
  }
}
class C7 {
  val  name1 = P1.O1.O2.name      // Okay  - 路径表达式指向某一字段
  type C1    = P1.O1.C1           // Okay  - 路径表达式指向某一“叶子”类
  val  c1    = new P1.O1.C1       // Okay  - 路径表达式指向某一“叶子”类
  // val name2 = P1.O1.C1.name    // ERROR - P1.O1.C1并不固定
}
```

C7 类中的成员 name1、C1 以及 c1 的路径表达式中除了最后一位之外，其余元素都是固定元素。而 name2 所使用的路径表达式中，在最后一个位置之前便出现了非固定元素（C1）。

删除 name2 声明前的注释符号之后，你便能看到下面这个编译错误：

```
[error] .../typepaths/path-expressions.scala:52: value C1 is not a member of
```

```
  object progscala2.typesystem.typepaths.P1.O1
[error]   val name2 = P1.O1.C1.name
[error]                     ^
```

如果能在代码中避免使用复杂路径，当然也不失为一个好的方法。

## 15.2 依赖方法类型

依赖方法类型（dependent method type）是一种路径相关类型的形式，它有助于解决许多设计问题，Scala 2.10 引入了这一新特性。

Magnet 设计模式（magnet 是“磁石”的意思）便应用了依赖方法类型，该设计模式中存在一个接受单一对象输入的处理方法，而这个输入对象便称为 magnet，它能够确保函数返回类型是可兼容的。如果想对这项技术的相关示例有更深的了解，请参考“spray.io 博客上的内容”（http://spray.io/blog/2012-12-13-the-magnet-pattern/）。我们也将通过一个示例对此进行讲解：

```
// src/main/scala/progscala2/typesystem/dependentmethodtypes/dep-method.sc


import scala.concurrent.{Await, Future}                  // ❶
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global

case class LocalResponse(statusCode: Int)                 // ❷
case class RemoteResponse(message: String)
...
```

❶ 导入 `scala.concurrent.Future`（http://www.scala-lang.org/api/current/scala/concurrent/Future.html）类和与异步计算相关的类。

❷ 定义了两个 case 类，用于返回计算后得出的“回复信息”，这些回复可能来自本地调用（调用者和被调用者在一个进程内），也可能来自远程服务调用。需要注意这两个 case 类并未共享公共父类，两者毫无关联。

在 17.2 节中，我们将对 `Future`（http://www.scala-lang.org/api/current/scala/concurrent/Future.html）类型进行深入讲解。而现在，我们只对需要了解的部分进行讲解：

```
sealed trait Computation {
  type Response
  val work: Future[Response]
}

case class LocalComputation(
    work: Future[LocalResponse]) extends Computation {
  type Response = LocalResponse
}
case class RemoteComputation(
    work: Future[RemoteResponse]) extends Computation {
  type Response = RemoteResponse
}
...
```

以 Computation 为父节点的这组密封类（sealed class）提供了服务所需要的所有计算类型：包括了本地处理和远程处理。由于所需要执行的工作也封装到一个 Future 对象中，因此这些计算将以异步的方式执行。本地处理将返回对应的 LocalResposne 对象，而远程处理则返回对应的 RemoteResponse 对象：

```
object Service {
  def handle(computation: Computation): computation.Response = {
    val duration = Duration(2, SECONDS)
    Await.result(computation.work, duration)
  }
}

Service.handle(LocalComputation(Future(LocalResponse(0))))
// Result: LocalResponse = LocalResponse(0)
Service.handle(RemoteComputation(Future(RemoteResponse("remote call"))))
// Result: RemoteResponse = RemoteResponse(remote call)
```

最后，我们定义了一个服务对象，该对象提供了单一入口点 handle 方法。而 handle 方法则通过 scala.concurrent.Await（http://www.scala-lang.org/api/current/scala/concurrent/Await$.html）对象等待 Future 对象执行完毕。Await.result 根据输入的 Computation 对象的具体类型，返回相应的 LocalResponse 对象或 RemoteResponse 对象。

请留意，由于 LocalResponse 与 RemoteResponse 毫不相关，因此 handle 方法并未返回它们共有的某个父类实例。handle 方法另辟蹊径，它将依据输入参数决定返回类型。由于无法通过类型检查的缘故，当输入参数类型为 RemoteComputation 时，不可能返回 LocalResponse，反之亦然。

## 15.3 类型投影

我们将回顾之前 15.1 节中讨论过的 Service 设计问题。首先，我们对 Service 类进行重写，从中抽取出一些可能在真实应用中更为典型的抽象：

```
// src/main/scala/progscala2/typesystem/valuetypes/type-projection.scala
package progscala2.typesystem.valuetypes

trait Logger {                                                  // ❶
  def log(message: String): Unit
}

class ConsoleLogger extends Logger {                            // ❷
  def log(message: String): Unit = println(s"log: $message")
}

trait Service {                                                 // ❸
  type Log <: Logger
  val logger: Log
}

class Service1 extends Service {                                // ❹
  type Log = ConsoleLogger
```

```
  val logger: ConsoleLogger = new ConsoleLogger
}
```

❶ 定义 Logger 特征。

❷ 具体的 Logger 类，为了简化，该类将日志输出到控制台中。

❸ Service 特征定义了抽象类型 Log，该类型是 Logger 类型的别名。与此同时，我们在 Service 特征中声明了一个 Log 类型的字段。

❹ 定义了具体的服务类，该类将使用 ConsoleLogger 打印输出日志。

假设我们希望“重用”Service1 中定义的 Log 类型。下面，我们将在 REPL 会话中尝试一些可能的重用方案：

```
// src/main/scala/progscala2/typesystem/valuetypes/type-projection.sc

scala> import progscala2.typesystem.valuetypes._

scala> val l1: Service.Log   = new ConsoleLogger
<console>:10: error: not found: value Service
       val l1: Service.Log   = new ConsoleLogger
               ^

scala> val l2: Service1.Log  = new ConsoleLogger
<console>:10: error: not found: value Service1
       val l2: Service1.Log  = new ConsoleLogger
               ^

scala> val l3: Service#Log   = new ConsoleLogger
<console>:10: error: type mismatch;
 found   : progscala2.typesystem.valuetypes.ConsoleLogger
 required: progscala2.typesystem.valuetypes.Service#Log
       val l3: Service#Log   = new ConsoleLogger
                               ^

scala> val l4: Service1#Log  = new ConsoleLogger
l4: progscala2.typesystem.valuetypes.ConsoleLogger =
  progscala2.typesystem.valuetypes.ConsoleLogger@6376f152
```

调用 Service.Log 和 Service1.Log 时，Scala 将分别查找名为 Service 和 Service1 的对象，但是这些伴生对象却并不存在。

不过，我们可以使用 # 符号来对我们想查找的类型进行映射。第一次尝试未能通过类型检查。尽管 Service.Log 和 ConsoleLogger 类型均为 Logger 类型的子类型，但是由于 Service.Log 是抽象类型，因此我们尚且不知道该类型是否确实是 ConsoleLogger 类型的子类型。换言之，最后出现的 Service.Log 的具体定义可能是 Logger 类的另外一个子类，且该子类与 ConsoleLogger 不兼容。

由于以静态的方式通过类型检查，因此只有 val l4 = Service1#Log = new ConsoleLogger 能够正确执行。

最后提一句，我们日常工作中编写的那些简单的类型名称为类型标识符（type designator）。这些类型标识符本质上是这些类型映射的缩写。下面列举了一些类型标识符和对应的类型

映射，该示例改编自《Scala 语言规范》的 3.2 节：

```
Int             // scala.type#Int
scala.Int       // scala.type#Int
package pkg {
  class MyClass {
    type t      // pkg.MyClass.type#t
  }
}
```

## 单例类型

我们已经对单例对象（singleton object）进行了学习，声明单例对象时需要使用关键字 object。Scala 中同样存在着单例类型的概念。AnyRef 子类的所有实例 v，包括 null，都对应了一个唯一的单例类型（singleton type）。我们可以使用 v.type 表达式得到实例 v 的单例类型。在对象声明体中使用该表达式，能够将允许的实例数量限定到 1 个。我们将沿用之前 Logger 和 Service 类型的示例为大家进行讲解：

```
// src/main/scala/progscala2/typesystem/valuetypes/type-types.sc

scala> val s11 = new Service1
scala> val s12 = new Service1

scala> val l1: Logger = s11.logger
l1: ...valuetypes.Logger = ...valuetypes.ConsoleLogger@3662093

scala> val l2: Logger = s12.logger
l2: ...valuetypes.Logger = ...valuetypes.ConsoleLogger@411c6639

scala> val l11: s11.logger.type = s11.logger
l11: s11.logger.type = progscala2.typesystem.valuetypes.ConsoleLogger@3662093

scala> val l12: s11.logger.type = s12.logger
<console>:12: error: type mismatch;
 found   : s12.logger.type (with underlying type ...valuetypes.ConsoleLogger)
 required: s11.logger.type
       val l12: s11.logger.type = s12.logger
                                      ^
```

我们只能使用 s11.logger 为 l11 和 l12 赋值。s12.logger 的类型并不与 s11 和 s12 的类型兼容。

定义单例对象时，同时定义了对象实例和其对应的类型：

```
// src/main/scala/progscala2/typesystem/valuetypes/object-types.sc

case object Foo { override def toString = "Foo says Hello!" }
```

如果你希望定义输入参数为该类型的方法，那么输入类型应设置为 Foo.type。

```
scala> def printFoo(foo: Foo.type) = println(foo)
printFoo: (foo: Foo.type)Unit

scala> printFoo(Foo)
```

```
Foo says Hello!
```

## 15.4 值的类型

每一个值都具有类型。“值类型”指的是所有可能的类型形式，这些类型格式我们曾接都接触过。

> 在本节中，我们使用的值类型（value type）一词与《Scala 语言规范》中的定义相同，不过，在本书的其他章节中，值类型指的是 AnyVal 的所有子类型，这也是值类型更为常用的意思。

值类型包括：参数类型、单例类型、类型映射、类型标识符、复合类型、既存类型、元组类型、函数类型以及中缀类型。除了常规的写法之外，Scala 还为最后三种类型提供了更为方便的简写语法，因此我们会对这三种类型进行回顾。在对这几种类型进行讲解的同时，我们会讲述一些之前未谈及的具体细节。

### 15.4.1 元组类型

我们都知道，Scala 允许我们通过 (A,B,C) 表达式声明 Tuple3[A,B,C] 对象，而这一类型也称为*元组类型*（tuple type）：

```
val t1: Tuple3[String, Int, Double] = ("one", 2, 3.14)
val t2: (String, Int, Double)       = ("one", 2, 3.14)
```

对于包含了复杂类型的元组而言，使用这种简化的写法会减少嵌套括号的数量，因此会更便利一些。除此之外，简化的写法中省略了 TupleN 字符，因而比之前的写法要简短一些。事实上，很少会有人使用 TupleN 格式的函数签名。请对比 List[Tuple2[Int,String]] 与 List[(Int,String)] 类型。

### 15.4.2 函数类型

我们可以使用箭头表达式编写类型为函数类型的对象，如 Function2 类型：

```
val f1: Function2[Int,Double,String] = (i,d) => s"int $i, double $d"
val f2: (Int,Double) => String = (i,d) => s"int $i, double $d"
```

在指定元组对象时，通常并不会使用 TupleN 这类复杂的语法。与之类似，我们也很少使用 FunctionN 来创建函数类型。

### 15.4.3 中缀类型

我们可以使用中缀表达式编写包含两个类型参数的类型。下面的示例使用了 Either[A,B] 类型：

```
val left1:  Either[String,Int] = Left("hello")
val left2:  String Either Int  = Left("hello")
val right1: Either[String,Int] = Right(1)
```

```
val right2: String Either Int  = Right(2)
```

你可以嵌套使用多个中缀类型。除了类型名称以冒号 (:) 结尾的情况，中缀类型是左结合的。假如类型名称以冒号结尾，该类型则为右结合。这与 for 术语的用法相同（之前没有强调过这点，这里我们称那些非类型表达式为术语，for 术语其实就是 for 表达式）。你可以使用小括号改写默认的类型结合顺序。

```
// src/main/scala/progscala2/typesystem/valuetypes/infix-types.sc

scala> val xll1:  Int Either Double  Either String  = Left(Left(1))
xll1: Either[Either[Int,Double],String] = Left(Left(1))

scala> val xll2: (Int Either Double) Either String  = Left(Left(1))
xll2: Either[Either[Int,Double],String] = Left(Left(1))


scala> val xlr1:  Int Either Double  Either String  = Left(Right(3.14))
xlr1: Either[Either[Int,Double],String] = Left(Right(3.14))

scala> val xlr2: (Int Either Double) Either String  = Left(Right(3.14))
xlr2: Either[Either[Int,Double],String] = Left(Right(3.14))


scala> val xr1:   Int Either Double  Either String  = Right("foo")
xr1: Either[Either[Int,Double],String] = Right(foo)

scala> val xr2:  (Int Either Double) Either String  = Right("foo")
xr2: Either[Either[Int,Double],String] = Right(foo)


scala> val xl:   Int Either (Double Either String)  = Left(1)
xl: Either[Int,Either[Double,String]] = Left(1)

scala> val xrl:  Int Either (Double Either String)  = Right(Left(3.14))
xrl: Either[Int,Either[Double,String]] = Right(Left(3.14))

scala> val xrr:  Int Either (Double Either String)  = Right(Right("bar"))
xrr: Either[Int,Either[Double,String]] = Right(Right(bar))
```

很显然，叠加中缀类型会使代码立刻变得复杂起来。

下面，我们将转入一个重要的、庞大的、有时候又很有挑战性的主题：higher-kinded 类型。

## 15.5　Higher-Kinded类型

操作 Seq 实例时，我们通常习惯编写下面的代码：

```
def sum(seq: Seq[Int]): Int = seq reduce (_ + _)

sum(Vector(1,2,3,4,5))   // 结果值：15
```

首先，我们将使用*类型类*（type class）对加法进行泛化（请回顾 5.4 节的内容）。通过使用类型类，我们能够对元素类型操作进行归纳：

```
// src/main/scala/progscala2/typesystem/higherkinded/Add.scala
package progscala2.typesystem.higherkinded

trait Add[T] {                                                   // ❶
  def add(t1: T, T2: T): T
}

object Add {                                                     // ❷
  implicit val addInt = new Add[Int] {
    def add(i1: Int, i2: Int): Int = i1 + i2
  }

  implicit val addIntIntPair = new Add[(Int,Int)] {
    def add(p1: (Int,Int), p2: (Int,Int)): (Int,Int) =
      (p1._1 + p2._1, p1._2 + p2._2)
  }
}
```

❶ Add 特征中定义了加法。

❷ Add 特征的伴生对象中定义了该 trait 的两个实例，这两个实例将作为 Int 类型和 Int 对的隐式值。

现在，我们尝试使用这两个隐式值：

```
// src/main/scala/progscala2/typesystem/higherkinded/add-seq.sc
import progscala2.typesystem.higherkinded.Add         // ❶
import progscala2.typesystem.higherkinded.Add._

def sumSeq[T : Add](seq: Seq[T]): T =                  // ❷
  seq reduce (implicitly[Add[T]].add(_,_))

sumSeq(Vector(1 -> 10, 2 -> 20, 3 -> 30))              // 结果值: (6,60)
sumSeq(1 to 10)                                        // 结果值: 55
sumSeq(Option(2))                                      // ❸ 出错!
```

❶ 导入 Add 特征以及在 Add 伴生对象定义的隐式对象。

❷ 通过使用上下文边界（context bound）和 implicitly 关键字（请参考 5.1 节），我们计算出了一组数据的总和。

❸ 由于 Option 并不是 Seq 类型的子类型，因此传入 Option 参数会导致程序出错。

对于任何一种序列，只要我们为它定义了隐式的 Add 实例，那么 sumSeq 方法便能计算出该序列的总和。

不过，sumSeq 仍然只支持 Seq 子类型。假如容器类型并不是 Seq 子类型，但是却实现了 reduce 方法，我们该如何对该容器进行处理呢？我们会使用更加泛化的求和操作。

Scala 支持 higher-kinded 类型，通过应用该类型，我们可以对参数化类型进行抽象。下面列举了该类型的一种使用方式：

```
// src/main/scala/progscala2/typesystem/higherkinded/Reduce.scala
package progscala2.typesystem.higherkinded
import scala.language.higherKinds                                // ❶
```

```
trait Reduce[T, -M[T]] {                                    // ❷
  def reduce(m: M[T])(f: (T, T) => T): T
}

object Reduce {                                             // ❸
  implicit def seqReduce[T] = new Reduce[T, Seq] {
    def reduce(seq: Seq[T])(f: (T, T) => T): T = seq reduce f
  }

  implicit def optionReduce[T] = new Reduce[T, Option] {
    def reduce(opt: Option[T])(f: (T, T) => T): T = opt reduce f
  }
}
```

❶ higher-kinded 类型属于可选功能。假如你未导入该功能，Scala 会抛出警告。

❷ Reduce 特征对“规约”操作（reduction）进行抽象，将其封装为 higher-kinded 类型 M[T]。在一些库中，把 higher-kinded 类型命名为 M 是不成文的惯例。

❸ 为了能对 Seq 和 Option 值执行规约操作，我们分别为这两类类型提供了隐式实例。为了简化起见，我们将直接使用类型中已经提供的 reduce 方法执行规约操作。

Reduce 特征的声明体中使用了 M[T] 逆变（contravariant，声明逆变时需要在 M[T] 前添加 - 符号），这样做的原因是什么呢？ 假如我们未在 M[T] 前添加 + 或 - 符号，那些作用于 Seq 类型的隐式实例将无法对 Seq 类型的子类型起作用，如 Vector 类型。（请尝试移除 – 符号，并运行之后的示例。）请注意，reduce 方法中传入的参数类型是 M[T] 的容器类型。正如我们在 10.1.1 节和 14.2.2 节中看到的那样，这些方法中的某些参数位于“逆变位”（这些参数所在的上下文，决定了这些参数应该可逆变）。因此，我们要求 Reduce 对 M[T] 而言是可逆变的。

与 Add 对象中定义的隐式值相比，seqReduce 和 optionReduce 都是隐式方法，而不是隐式值。这是因为我们需要从具体的实例中推导出类型参数 T。我们无法像 Add 对象那样将 seqReduce 和 optionReduce 定义为隐式 val 类型。

请注意，在 seqReduce[T] = new Reduce[T, Seq] {...} 表达式中，我们并未设置 Seq 类型的类型参数。该类型参数将从 Reduce 的定义中推导出来。假如你直接设置了 Seq 的类型参数，例如 new Reduce[T, Seq[T]]，你会得到这样一条让人困惑的错误信息 Seq[T] takes no type parameters, expected: one。

我们将使用 sum2 方法对 Option 实例和 Seq 实例执行规约操作：

```
// src/main/scala/progscala2/typesystem/higherkinded/add.sc
import scala.language.higherKinds
import progscala2.typesystem.higherkinded.{Add, Reduce}    // ❶
import progscala2.typesystem.higherkinded.Add._
import progscala2.typesystem.higherkinded.Reduce._

def sum[T : Add, M[T]](container: M[T])(                   // ❷
  implicit red: Reduce[T,M]): T =
    red.reduce(container)(implicitly[Add[T]].add(_,_))

sum(Vector(1 -> 10, 2 -> 20, 3 -> 30))                     // 结果值：(6,60)
```

```
sum(1 to 10)                                         // 结果值: 55
sum(Option(2))                                       // 结果值: 2
sum[Int,Option](None)                                // ❸ 错误!
```

❶ 导入 Add 特征和 Reduce 特征，之后导入这两个特征对应的伴生对象中定义的各类隐式。

❷ 定义了 sum 方法，该方法能够处理 higher-kinded 类型（我们将会对此进行详细说明）。

❸ 当对空容器执行 sum 操作（规约操作）时，将会出现错误。我们为 sum 方法添加的类型签名 [Int, Opton] 会要求编译器将 None 解释成 Option[Int] 类型。假如不添加该类型签名，我们将得到编译错误：无法判断 Option[T] 类型中的类型参数 T 到底应该对应 addInt 方法还是 addIntIntPair 方法。通过显式地指定类型，我们能够得到真正希望捕获的运行错误：我们不能对 None 值调用 reduce 方法（适用于所有空容器）。

sum 方法的实现并非没有意义。与之前一样，我们定义了上下文边界 T: Add。我们也希望定义 M[T] 的上下文边界，如 M[T] : Reduce。不过我们无法做到这点，因为 Reduce 特征包含两个类型参数，而上下文边界只适用于包含单参数的场景。因此，我们可以为 sum 方法添加第二个参数列表。该参数列表中包含一个隐式的 Reduce 参数，使用该参数可以对输入的集合执行 reduce 操作。

为进一步简化实现，我们可以重新定义 Reduce 特征，新定义的 Reduce 特征只包含一个类型参数且属于 higher-kinded 类型。这样一来，我们便能实现在上下文边界中重新使用 Reduce 特征：

```
// src/main/scala/progscala2/typesystem/higherkinded/Reduce1.scala
package progscala2.typesystem.higherkinded
import scala.language.higherKinds

trait Reduce1[-M[_]] {                                         // ❶
  def reduce[T](m: M[T])(f: (T, T) => T): T
}

object Reduce1 {                                               // ❷
  implicit val seqReduce = new Reduce1[Seq] {
    def reduce[T](seq: Seq[T])(f: (T, T) => T): T = seq reduce f
  }

  implicit val optionReduce = new Reduce1[Option] {
    def reduce[T](opt: Option[T])(f: (T, T) => T): T = opt reduce f
  }
}
```

❶ Reduce1 抽象体只包含一个类型参数 M，尽管 M 仍然是可逆变类型，但是我们未指定 M 使用的类型参数。因此，Reduce1 属于*既存类型*（existential type，请参考 14.9 节的相关内容）。而 T 参数也被移至 reduce 方法中。

❷ seqReduce 和 optionReduce 不再是方法，它们被声明为隐式值。

在之前的示例中，我们需要使用隐式方法，使得 Scala 能够推导出类型参数 T 的值，而现在我们仅仅定义了一些隐式实例，因此在调用 reduce 方法之前，Scala 无法推导出 T 的值。

修改后的 sum 方法也变得更加简洁，调用 sum 方法将返回相同的结果（我们将不再列出

sum 方法的结果）：

```
// src/main/scala/progscala2/typesystem/higherkinded/add1.sc
...

def sum[T : Add, M[_] : Reduce1](container: M[T]): T =
    implicitly[Reduce1[M]].reduce(container)(implicitly[Add[T]].add(_,_))
```

现在，我们定义了两个上下文边界，它们分别作用于 Reduce1 和 Add 特征。而使用 implicity 修饰的类型参数则能够区分出这两种不同的隐式值。

实际上，你看到的大多数 higher-kinded 类型都与本示例相似，它们都适用 M[_] 类型，而非 M[T]。

higher-kinded 类型给我们带来一些额外的抽象，并且使代码变得更加复杂，我们还应该使用这一 trait 吗？为了能够以一种非常简洁和强大的方式将代码组合起来，Scalaz（https://github.com/scalaz/scalaz）和 Shapeless（https://github.com/milessabin/shapeless）这样的类库大量使用了 higher-kinded 类型带来的额外抽象和复杂代码。不过，你需要考虑团队成员的开发能力。请对这点保持警觉，如果代码过于抽象，那么学习、测试、调试以及代码改进都会非常困难。

## 15.6 类型Lambda

类型 Lambda 指的是嵌入在另一函数中的函数，它只作用于类型级。假如我们需要使用的参数化类型中包含了太多的类型参数，便可以使用类型 Lambda 进行处理。类型 Lambda 是一个编程术语，它并不是类型系统的特殊功能。

下面这个示例了使用 map 方法，该示例与上一节中 reduce 示例的实现方法略有不同：

```
// src/main/scala/progscala2/typesystem/typelambdas/Functor.scala
package progscala2.typesystem.typelambdas
import scala.language.higherKinds

trait Functor[A,+M[_]] {                                              // ❶
  def map2[B](f: A => B): M[B]
}
object Functor {                                                      // ❷
  implicit class SeqFunctor[A](seq: Seq[A]) extends Functor[A,Seq] {
    def map2[B](f: A => B): Seq[B] = seq map f
  }
  implicit class OptionFunctor[A](opt: Option[A]) extends Functor[A,Option] {
    def map2[B](f: A => B): Option[B] = opt map f
  }

  implicit class MapFunctor[K,V1](mapKV1: Map[K,V1])                  // ❸
    extends Functor[V1,({type λ[α] = Map[K,α]})#λ] {                  // ❹
      def map2[V2](f: V1 => V2): Map[K,V2] = mapKV1 map {
        case (k,v) => (k,f(v))
      }
    }
}
```

❶ Functor 一词常用于对包含了 `map` 操作的类型进行命名。我们将在第 16 章“Functor 的种类”一节中对此进行解释。不同于我们之前见到的 `Reduce` 类型，`Functor` 类型包含的方法中并未将容器作为参数传入。我们将为提供了 `map2` 方法的 `Functor` 类定义隐式转换。之所以将方法命名为 map2 是因为这样一来便不会与通常的 `map` 方法混淆。这也意味着我们不再需要 `M[T]` 是可逆变的，事实上，将 `M[T]` 设置为协变类型有助于编写代码。

❷ 我们按照常用的方法为 `Seq` 和 `Option` 类型定义了隐式转换。为了简化起见，在 `map2` 的实现体中，我们将使用这两个类型自带的 `map` 方法。由于 `Functor` 对 `M[T]` 而言是可协变的，因此为 `Seq` 类型定义的隐式转换也适用于 `Seq` 类型的所有子类型。

❸ 该行代码是本示例的核心：定义了 `Map` 类型的转换，在该转换中，我们使用了两个类型参数，而不是一个。

❹ 使用类型 Lambda 对额外的类型参数进行处理。

在 `MapFunctor` 类中，我们“设定”对 `Map` 对象执行 map 操作时，`Map` 对象的 key 值保持不变，而 value 值则会被修改。而实际的 `Map.map` 方法（http://www.scala-lang.org/api/current/#scala.collection.Map）更为通用，它允许用户同时修改 key 值和 value 值（示例中的 `map` 方法实际上实现了 `Map.mapValues` 方法，http://www.scala-lang.org/api/current/#scala.collection.Map）。类型 Lambda 的语法有一些冗长，这使得开发人员很难立刻理解相关代码。下面我们将对其进行扩展，使其更易于理解：

```
... Functor[V1,                  // ❶
  (                              // ❷
    {                            // ❸
      type λ[α] = Map[K,α]       // ❹
    }                            // ❺
  )#λ                            // ❻
]
```

❶ `V1` 是 `Functor` 的第一个类型参数，而 `Functor` 预期第二个类型参数会是包含了类型参数的容器类型。

❷ 表达式（结束于第六行）对应的左括号。该左括号开启了第二个类型参数的定义。

❸ 开始定义结构化类型（请查看 14.7 节的相关内容）。

❹ 定义了类型成员 λ，该类型成员是 `Map` 类型的别名。尽管我们随意将该类型成员命名为 λ（λ 通常用于表示类型成员），不过由于 λ 恰巧与 Lambda 相匹配[1]，因此这个名字得到了很广泛的使用。λ 本身包含了一个类型参数 α（α 也是随意取的名字），在本示例中 α 表示了 `Map` 所使用的 key 类型。

❺ 结束结构化类型定义。

❻ 结束始于第二行的表达式，与此同时，运用类型映射机制将类型 λ 从参数化类型中取出（请回顾 15.3 节的内容）。λ 是带有嵌入类型参数的 `Map` 的别名，而 Scala 会通过后续的代码推导出嵌入的类型参数。

注 1：当然，如果使用 ASCII 字符进行命名，比如说 L，我们更容易在大多数键盘上操作。

由此我们可以发现，运用类型 Lambda 可以处理 `Map` 所需要的额外类型参数，而 `Functor` 则不支持这一功能。由于 Scala 会根据后面的代码推导出 α 值，因此我们并不需要显式地对 λ 和 α 进行重复引用。

下面的脚本证明了上述代码可以成功地执行：

```
// src/main/scala/progscala2/typesystem/typelambdas/Functor.sc
import scala.language.higherKinds
import progscala2.typesystem.typelambdas.Functor._

List(1,2,3) map2 (_ * 2)              // List(2, 4, 6)
Option(2) map2 (_ * 2)                // Some(4)
val m = Map("one" -> 1, "two" -> 2, "three" -> 3)
m map2 (_ * 2)                        // Map(one -> 2, two -> 4, three -> 6)
```

你无需经常使用类型 Lambda 的语法，不过它有助于解决描述的问题。Scala 的未来版本也许会为类型 Lambda 习语提供更简洁的语法。

## 15.7　自递归类型：F-Bounded多态

从技术角度讲，自递归类型也被称为 F-bounded 多态类型，用于表示指向自身的类型。Java 的 Enum（http://docs.oracle.com/javase/8/docs/api/java/lang/Enum.html）抽象类便是经典的一例，也是所有 Java 枚举的基础类型，其声明如下：

```
public abstract class Enum<E extends Enum<E>>
extends Object
implements Comparable<E>, Serializable
```

大多数 Java 开发者都会对 `Enum<E extends Enum<E>>` 这样的语法感到困扰，不过该语法却能带来一些重要的好处。`Comparable<E>` 接口中声明的 `compareTo` 方法也使用了这类语法：

```
int compareTo(E obj)
```

如果传入 `compareTo` 方法的对象并不是相同类型的某一枚举值，便会出现编译错误。下面我们将使用 JDK 中 `Enum` 类型的两个子类型：`java.util.concurrent.TimeUnit`（http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TimeUnit.html）和 `java.net.proxy.Type`（http://docs.oracle.com/javase/8/docs/api/java/net/Proxy.Type.html）进行演示（将省略某些细节）：

```
scala> import java.util.concurrent.TimeUnit
scala> import java.net.Proxy.Type

scala> TimeUnit.MILLISECONDS compareTo TimeUnit.SECONDS
res0: Int = -1

scala> Type.HTTP compareTo Type.SOCKS
res1: Int = -1

scala> TimeUnit.MILLISECONDS compareTo Type.HTTP
<console>:11: error: type mismatch;
 found   : java.net.Proxy.Type(HTTP)
```

```
  required: java.util.concurrent.TimeUnit
              TimeUnit.MILLISECONDS compareTo Type.HTTP
                                                ^
```

在 Scala 中，使用递归类型能够方便我们定义返回类型与调用者类型相同的方法，即便是返回类型与调用者类型具有继承关系，递归类型也能起作用。在下面的示例中，make 方法应该返回与调用者类型相同的实例，而不是声明 make 方法的 Parent 类型。

```
// src/main/scala/progscala2/typesystem/recursivetypes/f-bound.sc

trait Parent[T <: Parent[T]] {                                    // ❶
  def make: T
}

case class Child1(s: String) extends Parent[Child1] {             // ❷
  def make: Child1 = Child1(s"Child1: make: $s")
}

case class Child2(s: String) extends Parent[Child2] {
  def make: Child2 = Child2(s"Child2: make: $s")
}

val c1  = Child1("c1")           // c1: Child1 = Child1(c1)
val c2  = Child2("c2")           // c2: Child2 = Child2(c2)
val c11 = c1.make                // c11: Child1 = Child1(Child1: make: c1)
val c22 = c2.make                // c22: Child2 = Child2(Child2: make: c2)

val p1: Parent[Child1] = c1      // p1: Parent[Child1] = Child1(c1)
val p2: Parent[Child2] = c2      // p2: Parent[Child2] = Child2(c2)
val p11 = p1.make                // p11: Child1 = Child1(Child1: make: c1)
val p22 = p2.make                // p22: Child2 = Child2(Child2: make: c2)
```

❶ Parent 特征中定义了递归类型。定义 Parent 特征的语法与我们之前看到的 Java 中声明的 Enum 的语法是等价的。

❷ 声明继承类型时必须使用 X extends Parent[X] 这样的签名体。

请注意变量的类型签名，该类型签名出现在示例脚本最后创建的变量值注释中。例如：p22 变量是 Child2 类型，即使我们调用 make 方法时传入了 Parent 对象引用，该变量仍为 Child2 类型。

# 15.8　本章回顾与下一章提要

Shapeless 库应当是最大程度地应用了类型系统的最好例子（https://github.com/milessabin/shapeless)。Scalaz 库中也广泛使用了一些高级类型概念。努力学习并掌握类型系统大有益处，它能够为你提供一些解决设计难题的创新工具。

请注意，你无须掌握 Scala 丰富的类型系统所提供的所有复杂点，便能很好的使用 Scala 语言。不过，你对 Scala 类型系统的具体细节了解的越多，就越容易应用那些使用了这些复杂点的第三方库。你也能够构造属于你自己的强大、复杂的类库。

下一章中，我们将深入学习函数式编程的更为高阶的内容。

# 第16章

# 高级函数式编程

让我们回到函数式编程，并讨论一些高级概念。如果你是一位初学者，你可以跳过这一章。但如果你接触过诸如*代数数据类型*（algebraic data type）、*范畴理论*（category theory）和*单子*（monad）这样的术语，你需要再回过头来看这一章。

本章的目标是在不深入过多理论和符号的前提下，帮助你认识到这些概念是什么以及它们为什么如此重要。

## 16.1 代数数据类型

*抽象数据类型*（abstract data type）和*代数数据类型*（algebraic data type）都常常以 ADT 的形式进行缩写，这令人很困惑。前者在面向对象编程中很常见，它包括我们熟悉的 `Seq`，`Seq` 是所有序列容器的抽象。相对地，代数数据类型来源于函数式编程，你可能不太熟悉，但它同样重要。

代数数据类型这一术语的产生是由于我们将要讨论的很多种数据类型符合代数特性，也就是数学特性。这一点非常重要，因为如果能够证明类型的属性，我们就有信心认为它们是没有 bug 的，并且可以通过组合安全地构建起更复杂的数据结构和算法。

### 16.1.1 加法类型与乘法类型

Scala 类型分为*加法类型*（sum type）与*乘法类型*（product type）。

大部分你已知的类型都是乘法类型。比如说，当定义一个 case 类时，你可以拥有多少个独一无二的实例呢？考虑下面这个简单的例子：

```
case class Person(name: Name, age: Age)
```

你可以拥有的 Person 实例的个数等于 Name 实例的个数乘以 Age 的实例个数。比方说，Name 封装了非空字符串，并禁止非字母字符。这样，仍然会有无限多个有效值，但我们假设个数为 *N*。同样，Age 仅限于整数值，比方说介于 0 到 130 之间。但为什么不分别使用 String 和 Int 来举例呢？因为我们一直在强调，只要有可能，类型就应该表明允许的合法状态，并防止无效状态的出现。

由于我们可以用任意的 Name 值与任意的 Age 值组合起来创建 Pearson 值，所以 Person 可能的实例个数为 *131*N*。正因为如此，这样的类型被称为乘法类型。我们接触的大部分类型都属于这个范畴。

Scala 中 Product 类型的名称也源于此。Product 是所有 TupleN 和所有 case 类的父类，我们在 10.4 节中已经学习过。

在 2.6 节我们了解到 Unit 的单个实例有个神秘的名字——()。如果我们将它看做一个零元素的元组的话，这个古怪的名字其实是有道理的。而一个包含单个整数值的元组，即 (Int) 或 Tuple1[Int] 可以拥有 $2^{32}$ 个值，每个值对应一个整数值。一个没有元素的元组只能有一个实例，因为它不能携带任何状态。

试想，如果我们拥有包含两个元素的元组 (Int, String)，然后向元组追加 Unit，构造一个新的元组，看看会发生什么：

```
type unitTimesTuple2 = (Int, String, Unit)
```

这个类型具有多少个可能的实例？与类型 (Int, String) 拥有的实例个数相同。从乘法的角度看，这就像我们对实例个数乘以 1。所以，这就是 Unit 这个名字的来源，就像 1 是乘法的“单位”，0 是加法的单位一样。

乘法类型有零个实例的情况吗？我们需要一个包含零个实例的类型 scala.Nothing。将 Nothing 与任意其他类型组合，构造的新类型也只能包含零个实例，因为没有一个实例可以“装下”Nothing 字段。

加法类型的一个例子是枚举类型。回顾第 3 章的这个例子：

```
// src/main/scala/progscala2/rounding/enumeration.sc

object Breed extends Enumeration {
  type Breed = Value
  val doberman = Value("Doberman Pinscher")
  val yorkie   = Value("Yorkshire Terrier")
  val scottie  = Value("Scottish Terrier")
  val dane     = Value("Great Dane")
  val portie   = Value("Portuguese Water Dog")
}
```

这个类型有 5 个实例。需要注意的是这些值是相互排斥的。我们不能让它们进行组合（忽略狗生育的真实场景）。特定的品种有且只有一个。

实现加法类型的另一个方法是使用对象的封闭继承（sealed hierarchy of object）：

```
sealed trait Breed { val name: String }
case object doberman extends Breed { val name = "Doberman Pinscher" }
```

```
case object yorkie   extends Breed { val name = "Yorkshire Terrier" }
case object scottie  extends Breed { val name = "Scottish Terrier" }
case object dane     extends Breed { val name = "Great Dane" }
case object portie   extends Breed { val name = "Portuguese Water Dog" }
```

> 你只需要带索引的“标志位”时，使用枚举或对用户友好的字符串。如果需要携带更多的状态信息，那就使用对象的封闭继承。

### 16.1.2 代数数据类型的属性

在数学中，代数通过三个方面定义。

(1) 一系列对象：不要与 OO 语境中的对象混淆。这里说的对象可以是数字或任何东西。

(2) 一系列操作：表示元素如何组合在一起创建新元素。

(3) 一系列规则：定义了操作与对象之间的关系。例如：对于数组，存在以下规则：$(x + (y + z)) == ((x + y) + z)$（即结合律）。

下面我们先来考虑乘法类型。关于实例个数的非正式参数，现在用操作和数学规则加以正式描述。再次考虑将 `Unit`“加”到 `(Int, String)` 上的操作。它符合交换律：

```
Unit x (Int,String) == (Int,String) x Unit
```

从类型的实例个数的角度看，这是正确的。这就像任意 N 都满足 $1*N = N*1$ 一样。这一点可以推广到非 `Unit` 类型：

```
Breeds x (Int,String) == (Int,String) x Breeds
```

就像对任意的数字 $M$ 和 $N$ 都有 $M*N = N*M$ 一样。类似地，与“零”（`Nothing`）相乘也满足交换律：

```
Nothing x (Int,String) == (Int,String) x Nothing
```

谈到加法类型，我们应该记得集合的元素具有唯一性。因此，我们能想到，允许的合法犬种可以构成一个集合。这就意味着，向集合中加入 `Nothing`，依然得到同一个集合。而向集合中加入 `Unit` 则会创建一个新的集合，新集合中包括所有原来的元素再加上一个新元素 `Unit`。类似地，如果我们添加非 `Unit` 的新类型，新集合将包含该新类型所允许的所有实例，以及集合原有的元素。这与加法的代数规则相一致：

```
Nothing + (doberman, yorkie, ...) == (doberman, yorkie, ...) + Nothing
Unit    + (doberman, yorkie, ...) == (doberman, yorkie, ...) + Unit
Person  + (doberman, yorkie, ...) == (doberman, yorkie, ...) + Person
```

甚至还有一个形式为 $x*(a + b) = x*a + x*b$ 的分配律，请自行验证分配律的有效性。

### 16.1.3 代数数据类型的最后思考

还有更多有待探索的属性，不过我推荐你参阅 Chris Taylor 的博客系列（http://chris-taylor.github.io/blog/2013/02/10/the-algebra-of-algebraic-data-types/），他向我们出色地讲解了更多细节。

这一切与编程有什么关系呢？这种精确的推理鼓励我们审视我们的类型。这些类型是否有确切的含义？它们有没有将合法的值限制在那些有意义的值上？它们组合起来创建新类型时是否有精确的行为？

我最喜欢的有关“意想不到”的精确的例子是 Scala 中的 `List` 类型。它是一个抽象类，只有两个具体子类 `Nil` 和 `::`，后者表示非空列表。说一个 `List` 要么是空的，要么是非空的，这听起来好像很没有意义，但通过这两种类型我们可以构造出所有的列表，并精确界定列表应该有的行为。

## 16.2 范畴理论

也许在 Scala 社区中最具争议的辩论是多大程度上接受范畴理论。范畴理论是数学的一个分支，我认为这是函数式设计模式的来源。本节介绍范畴理论的基本思想和函数式编程最常用的几个实际范畴。它们是非常强大的工具，至少对于那些愿意掌握它们的开发团队来说是这样。

使用范畴理论有很大的争议，因为它的数学基础令人生畏。可访问的文档很难找到。范畴理论在全局属性的层面概括了所有的数学概念。因此，它提供了深刻而深远的抽象，但是当应用到代码中的时候，许多开发者都在同极度的抽象做斗争。拥有具体、特定细节的代码更容易为大多数人理解，但我们也知道抽象非常有用。问题的关键在于寻求平衡，尤其是在你感觉很舒服的位置打破平衡，这将决定范畴理论是否适合你。

然而，目前范畴理论在高级函数式编程中占有中心位置。它被率先用在 Haskell 中解决各种设计问题，并突破函数式思想的常规。现在大多数函数式语言都提供了常见类别的实现。

如果你是一名 Scala 的高级开发者，你应该学习范畴理论的基本理论，并应用于实际编程，然后再决定它是否适合你的团队和项目。不幸的是，我发现在组织上范畴理论支持者写的库却失败了。这是因为团队的其他成员发现这个库太难理解和维护。如果你也支持范畴理论，一定要考虑所写代码的生命周期和社会发展的因素。

Scalaz（发音为 Scala Zed）是实现了范畴的主要 Scala 库，也是学习和实验的一个好载体。我们已经在 7.4.4 节中利用了其中的 `Validation` 类型。在本章中，我将尽量简化范畴的实现，以减少学习的成本。

从某种意义上说，本节内容是 15.5 节内容的延续。在 15.5 节，我们讨论了参数化类型的抽象。例如：如果有一个 `Seq[A]` 方法，我们是否可以将其推广到 `M[A]`？在这里 `M` 是一个类型参数，表示任意一个被参数化的类型。现在，我们将对函数式的概念，如组合子、`map`、`flatMap` 等进行抽象。

## 16.2.1　关于范畴

我们从范畴的一般定义开始，其定义包括三个“实体”（类似代数的定义）。

(1) 一个包含一系列对象的类别。这与 OOP 对应的术语不同，但含义类似。

(2) 一组态射（morphism），也称为箭头（arrow）。态射是函数概念的一种推广，写为 $f: A \rightarrow B$（即 Scala 中的 `f: A => B`）。对于每个态射 $f$，其中一个对象为域（domain），另一个为值域（codomain）。用对象这个词感觉有些奇怪，但在有些范畴中，每个对象本身就是一系列值或其他范畴的集合。

(3) 一个称为态射组合的二元操作，其特性是，对于 $f: A \rightarrow B$ 与 $g: B \rightarrow C$，其组合为 $g \circ f: A \rightarrow C$。

态射组合满足两个公理。

(1) 每个对象 $x$ 有且仅有一个单位态射。也就是说，当域和值域相同时，$ID_x$ 与单位态射的组合有以下属性：$f \circ ID_x = ID_x \circ f$。

(2) 结合律：对于 $f: A \rightarrow B$，$g: B \rightarrow C$，$h: C \rightarrow D$，存在 $(f \circ g) \circ h = f \circ (g \circ h)$。

接下来我们将讨论的范畴具备以上特性和规则。我们只研究两个（在众多数学范畴中）软件开发中用到的范畴，Functor 和 Monad。我们还会提到另外两个，即 Applicative 与 Arrow 范畴。

## 16.2.2　Functor范畴

Functor 抽象了映射（`map`）操作。我们曾在 15.6 节引入 `map` 以实现一个需要使用类型 Lambda 表达式的例子。但这里，我们将用一种稍微不同的方式来实现它。首先定义抽象，然后在三个具体类型 `Seq`、`Option` 和 `A => B` 中实现 `map`：

```
// src/main/scala/progscala2/fp/categories/Functor.scala
package progscala2.fp.categories
import scala.language.higherKinds

trait Functor[F[_]] {                                              // ❶
  def map[A, B](fa: F[A])(f: A => B): F[B]                         // ❷
}

object SeqF extends Functor[Seq] {                                 // ❸
  def map[A, B](seq: Seq[A])(f: A => B): Seq[B] = seq map f
}

object OptionF extends Functor[Option] {
  def map[A, B](opt: Option[A])(f: A => B): Option[B] = opt map f
}

object FunctionF {                                                 // ❹
  def map[A,A2,B](func: A => A2)(f: A2 => B): A => B = {           // ❺
    val functor = new Functor[({type λ[β] = A => β})#λ] {          // ❻
      def map[A3,B](func: A => A3)(f: A3 => B): A => B = (a: A) => f(func(a))
    }
```

```
    functor.map(func)(f)                                      // ❼
  }
}
```

❶ 相比 15.6 节中上一个版本的代码，这里的 map 方法的参数是 F 的实例（F 是某种类型的容器）。此外，像以前一样，我们不使用 map2 做名字。

❷ map 的参数是 F[A]，它是一个类型为 A => B 的函数。返回值是 F[B] 类型的实例。

❸ Seq 和 Option 的实现对象。

❹ 定义一个实现对象，用于将一个函数映射到另一个函数，这可没那么容易！

❺ FunctionF 定义了自己的 map 方法，对 map 的调用与 Seq、Option 和其他任意我们要实现的转换的语法相同。这个 map 方法的输入参数是我们要转换的函数和执行该转换的函数。注意看这里的类型：我们要将 A => A2 函数转为 A => B 函数，意味着 map 的第二个函数参数 f 的类型应该是 A2 => B。换句话说，我们是在将函数级联起来。

❻ 在 map 的实现中，用恰当的类型构造了一个 Functor，用于实现转换。

❼ 最后，FunctionF.map 调用该 Functor，其中 FunctionF.map 的返回值是 A => B。

FunctionF 是平凡的。但理解它时，应该记住我们并没有改变最初的类型 A，只是在后面又级联了另一个函数，该函数采用 func 的输出 A2 作为其输入值，然后调用 f。

下面我们来试试这些类型：

```
// src/main/scala/progscala2/fp/categories/Functor.sc
import progscala2.fp.categories._
import scala.language.higherKinds

val fii: Int => Int       = i => i * 2
val fid: Int => Double    = i => 2.1 * i
val fds: Double => String = d => d.toString

SeqF.map(List(1,2,3,4))(fii)              // Seq[Int]: List(2, 4, 6, 8)
SeqF.map(List.empty[Int])(fii)            // Seq[Int]: List()

OptionF.map(Some(2))(fii)                 // Option[Int]: Some(4)
OptionF.map(Option.empty[Int])(fii)       // Option[Int]: None

val fa = FunctionF.map(fid)(fds)                                // ❶
fa(2)                                     // String: 4.2

// val fb = FunctionF.map(fid)(fds)                                ❷
val fb = FunctionF.map[Int,Double,String](fid)(fds)
fb(2)

val fc = fds compose fid                                        // ❸
fc(2)                                     // String: 4.2
```

❶ 将 Int => Double 类型的函数与 Double => String 类型的函数级联在一起，创造一个新函数，然后在下一行调用这个函数。

❷ 不幸的是，在这个函数字面量 FunctionF.map 中，参数的类型无法推断，所以必须使用显式的类型。

❸ 请注意，`FunctionF.map(f1)(f2) == f2 compose f1`，而不是 `f1 compose f2`！

那么，为什么带 `map` 操作的参数化类型被称为 Functor 呢？让我们再看一下 `map` 的声明。为了简单起见，我们用 `Seq` 对其重新定义，并对参数列表进行转换：

```
scala> def map[A, B](seq: Seq[A])(f: A => B): Seq[B] = seq map f

scala> def map[A, B](f: A => B)(seq: Seq[A]): Seq[B] = seq map f
```

现在，当我们使用第二个版本的部分应用程序时，注意返回新函数的类型：

```
scala> val fm = map((i: Int) => i * 2.1) _
fm: Seq[Int] => Seq[Double] = <function1>
```

所以，这个 `map` 方法将函数 `A => B` 提升为了 `Seq[A] => Seq[B]`！一般情况下，对于所有的 `A` 和 `B` 类型，`Functor.map` 将 `A => B` 态射为 `F[A] => F[B]`，其中 `F` 必须是一个范畴。换句话说，Functor 允许我们对保存一个或多个 `A` 值的“上下文”应用纯函数（`f: A => B`）。而不需要我们自己将 `A` 值提取出来再对这些值应用函数 `f`，最后再将结果放进一个新的“上下文”中。Functor 一词用于抽象纯函数的这种用法。

在范畴理论中，其他的范畴都是对象，而态射是范畴间的映射。例如：`List[Int]` 和 `List[String]` 就是两个范畴，它们各自的对象是所有可能的 `Int` 值和 `String` 值组成的列表。

在范畴理论的一般特性和公理之外，`Functor` 还有两个属性。

(1) Functor `F` 能保持单位值。也就是说，域中的单位值映射到值域中，仍然是单位。

(2) Functor `F` 能保持组合 $F(f \circ g) = F(f) \circ F(g)$。

下面给出第一个属性的例子，空列表是列表的“单位”。想想看，当你把空列表与另一个列表连接会发生什么。对空列表的映射依然返回空列表，但列表元素类型可能变了。

这些一般性的或者 Functor 独有的属性是否真的满足呢？以下的 ScalaCheck 属性测试代码对其进行了验证：

```
// src/test/scala/progscala2/fp/categories/FunctorProperties.scala
package progscala2.fp.categories
import org.scalatest.FunSpec
import org.scalatest.prop.PropertyChecks

class FunctorProperties extends FunSpec with PropertyChecks {

  def id[A] = identity[A] _    // 将identity method提升为函数

  def testSeqMorphism(f2: Int => Int) = {                      // ❶
    val f1: Int => Int = _ * 2
    import SeqF._
    forAll { (l: List[Int]) =>
      assert( map(map(l)(f1))(f2) === map(l)(f2 compose f1) )
    }
  }
```

```
def testFunctionMorphism(f2: Int => Int) = {                      // ❷
  val f1: Int => Int = _ * 2
  import FunctionF._
  forAll { (i: Int) =>
    assert( map(f1)(f2)(i) === (f2 compose f1)(i) )              // ❸
  }
}

describe ("Functor morphism composition") {                        // ❹
  it ("works for Sequence Functors") {
    testSeqMorphism(_ + 3)
  }
  it ("works for Function Functors") {
    testFunctionMorphism(_ + 3)
  }
}

describe ("Functor identity composed with a another function commutes") {
  it ("works for Sequence Functors") {                             // ❺
    testSeqMorphism(id[Int])
  }
  it ("works for Function Functors") {
    testFunctionMorphism(id)
  }
}

describe ("Functor identity maps between the identities of the categories") {
  it ("works for Sequence Functors") {                             // ❻
    val f1: Int => String = _.toString
    import SeqF._
    assert( map(List.empty[Int])(f1) === List.empty[String] )
  }
  it ("works for Function Functors") {
    val f1: Int => Int = _ * 2
    def id[A] = identity[A] _    // 将方法提升为函数
    import FunctionF._
    forAll { (i: Int) =>
      assert( map(id[Int])(f1)(i) === (f1 compose id[Int])(i) )
    }
  }
}

describe ("Functor morphism composition is associative") {
  it ("works for Sequence Functors") {                             // ❼
    val f1: Int => Int = _ * 2
    val f2: Int => Int = _ + 3
    val f3: Int => Int = _ * 5
    import SeqF._
    forAll { (l: List[Int]) =>
      val m12 = map(map(l)(f1))(f2)
      val m23 = (seq: Seq[Int]) => map(map(seq)(f2))(f3)
      assert( map(m12)(f3) === m23(map(l)(f1)) )
    }
  }
  it ("works for Function Functors") {
```

```
      val f1: Int => Int = _ * 2
      val f2: Int => Int = _ + 3
      val f3: Int => Int = _ * 5
      val f:  Int => Int = _ + 21
      import FunctionF._
      val m12 = map(map(f)(f1))(f2)
      val m23 = (g: Int => Int) => map(map(g)(f2))(f3)
      forAll { (i: Int) =>
        assert( map(m12)(f3)(i) === m23(map(f)(f1))(i) )
      }
    }
  }
}
```

❶ 一个辅助函数，对 SeqF 验证态射组合。从本质上讲，先对 Functor 与一个函数做映射，然后将输出的结果与第二个函数做映射，这样产生的函数与先将函数进行组合，再执行一次映射产生的函数相同吗？

❷ 一个类似的辅助函数，对 FunctionF 验证态射的组合。

❸ 注意我们对函数做了“态射”，然后用一系列生成的 Int 值来验证函数是否返回相同的输出。

❹ 对 SeqF 和 FunctionF 同时验证态射组合。

❺ 对 SeqF 和 FunctionF 验证单位特性。

❻ 验证 Functor 独有的特性：单位经过映射后还是单位。

❼ 验证 Functor 独有的特性：态射满足交换律。

回到程序本身，是否有必要冒着给代码增加复杂性的风险，定义一个额外的 map 抽象，这具有实际意义吗？在一般情况下，对数学上可证明的性质进行抽象，有助于我们推理出程序的结构和行为。例如，一旦有了广义的抽象 map，我们就可以将其应用到许多不同的数据结构，甚至函数中。这种范畴理论的推理能力已经在计算机科学研究的多个领域得到应用。

### 16.2.3　Monad范畴

如果说 Functor 是对 map 的抽象，那么有没有与 flatMap 相对应的抽象呢？的确有，就是 Monad。该名称源于古希腊的毕达哥拉斯学派哲学家所创造的 monas 一词，翻译过来大致意思是“生成其他所有事物的神”。

以下是我们对 Monad 的定义：

```
// src/main/scala/progscala2/fp/categories/Monad.scala
package progscala2.fp.categories
import scala.language.higherKinds

trait Monad[M[_]] {                                              // ❶
  def flatMap[A, B](fa: M[A])(f: A => M[B]): M[B]                // ❷
  def unit[A](a: => A): M[A]                                     // ❸

  // 一些常用别名:                                                   ❹
```

```
  def bind[A,B](fa: M[A])(f: A => M[B]): M[B] = flatMap(fa)(f)
  def >>=[A,B](fa: M[A])(f: A => M[B]): M[B] = flatMap(fa)(f)
  def pure[A](a: => A): M[A] = unit(a)
  def `return`[A](a: => A): M[A] = unit(a)    // 添加反引号,避免与关键字冲突
}

object SeqM extends Monad[Seq] {
  def flatMap[A, B](seq: Seq[A])(f: A => Seq[B]): Seq[B] = seq flatMap f
  def unit[A](a: => A): Seq[A] = Seq(a)
}

object OptionM extends Monad[Option] {
  def flatMap[A, B](opt: Option[A])(f: A => Option[B]):Option[B]= opt flatMap f
  def unit[A](a: => A): Option[A] = Option(a)
}
```

❶ 用 M[_] 表示拥有“Monad 性质”的类型。跟 Functor 一样，它只带有一个类型参数。

❷ 注意传给 flatMap 的参数 f 类型为 A => M[B]，而不是 A => B。

❸ Monad 还有第二个函数，输入参数 a，它将在 Monad 实例中返回。在 Scala 中，这通常是由构造器和 case 类的 apply 方法实现的。

❹ 数学和编程语言使用不同的术语。这里的 >>= 和 return 是 Haskell 的标准。但在 Scala 中，这两个名字是有问题的。由于 = 操作符的优先级问题，在 >>= 结尾的 = 会带来有趣的行为。除非像代码所示的一样，将 return 用引号引起，否则这个名字会与关键字冲突。

有时对 flatMap，也就是 bind 的抽象，被称为 Bind。

更常见的是，只对 unit 或 pure（纯性）的抽象被称为 Applicative。注意 unit 与 case 类的 apply 方法多么相似，两者都传入了一个值，然后返回一个类型实例！作为对构造的抽象，Applicative 也非常有趣。回想 5.2.3 节和 12.3.2 节中 CanBuildFrom 在集合库是如何用来构造新的集合实例的。如果不用那么灵活，Applicative 可以作为另一种选择。

我们来尝试使用 Monad 的实现：

```
// src/main/scala/progscala2/fp/categories/Monad.sc
import progscala2.fp.categories._
import scala.language.higherKinds

val seqf: Int => Seq[Int] = i => 1 to i
val optf: Int => Option[Int] = i => Option(i + 1)

SeqM.flatMap(List(1,2,3))(seqf)           // Seq[Int]: List(1,1,2,1,2,3)
SeqM.flatMap(List.empty[Int])(seqf)       // Seq[Int]: List()

OptionM.flatMap(Some(2))(optf)            // Option[Int]: Some(3)
OptionM.flatMap(Option.empty[Int])(optf)  // Option[Int]: None
```

描述 flatMap 的一种方法是：它从左边的容器中提取类型 A 的一个元素，将其绑定到新容器实例中的一个新元素中，并由此得名。类似 Map，它不需要知道如何从 M[A] 中提取元素。不过，看起来它的函数参数必须知道如何构建一个新的 M[B]。实际上，这并不是问

题，因为调用 unit 可以做到这一点。在 Scala 这样的面向对象的编程语言中，实际返回的 Monad 类型可能是 M 的子类型。

Monad 的规则如下。

unit 的行为类似单位（正如其名）：

```
flatMap(unit(x))(f) == f(x)        这里的x是一个值
flatMap(m)(unit) == m              这里m是Monad的一个实例
```

类似 Functor 的态射组合，先后对两个函数做 flatMap，就像对两个函数的组合函数做一次 flatMap 一样：

```
flatMap(flatMap(m)(f))(g) == flatMap(m)(x => flatMap(f(x))(g))
```

代码示例包含属性测试，以验证这些属性（见 src/test/scala/progscala2/toolslibs/fp/MonadProperties.scala）。

### 16.2.4 Monad的重要性

讽刺的是，在范畴理论中 Functor 比 Monad 更重要；但在软件应用中，Funtor 的重要性则远比不上 Monad。

从本质上说，Monad 之所以重要，是因为它为我们提供了一个对某个值包装上下文信息的规范方法。当这个值发生变化时，Monad 可以传递给上下文并引起相关变化。这样，就可以将值和上下文之间的耦合降到最低。而 Monad 可以通知读者上下文的存在。

这种“模式”在 Scala 中很常用，该模式率先在 Haskell 中使用，之后启发了 Scala。我们在 7.4 节接触过一些例子，包括 Option、Either、Try 和 scalaz.Validation。

它们都满足 Monad 特性，因为它们都支持 flatMap 及构造（case 类的 apply 方法，而不是 unit）。它们允许操作序列，并可以用不同的方式对失败进行处理，通常是返回父类型的子类实例。

回想 flatMap 的简化版函数签名，其中使用了 Try：

```
sealed abstract class Try[+A] {
  def flatMap[B](f: A => Try[B]): Try[B]
}
```

其他类型也是类似的。接下来我们考虑多步骤的处理，其中上一步骤的处理结果是下一步骤的输入，遇到第一个失败时就停止处理：

```
// src/main/scala/progscala2/fp/categories/for-tries-steps.sc

import scala.util.{ Try, Success, Failure }

type Step = Int => Try[Int]                                    // ❶

val successfulSteps: Seq[Step] = List(                         // ❷
  (i:Int) => Success(i + 5),
  (i:Int) => Success(i + 10),
```

```
  (i:Int) => Success(i + 25))
val partiallySuccessfulSteps: Seq[Step] = List(
  (i:Int) => Success(i + 5),
  (i:Int) => Failure(new RuntimeException("FAIL!")),
  (i:Int) => Success(i + 25))

def sumCounts(countSteps: Seq[Step]): Try[Int] = {                 // ❸
  val zero: Try[Int] = Success(0)
  (countSteps foldLeft zero) {
    (sumTry, step) => sumTry flatMap (i => step(i))
  }
}

sumCounts(successfulSteps)
// 返回: scala.util.Try[Int] = Success(40)

sumCounts1(partiallySuccessfulSteps)
// 返回: scala.util.Try[Int] = Failure(java.lang.RuntimeException: FAIL!)
```

❶ “步骤”函数的别名。

❷ 两个包含多个步骤的序列，其中一个序列全部成功，另一个序列里有一个步骤会失败。

❸ 一个方法，输入一个步骤序列，将每个步骤的输出输入到下一个步骤去执行。

sumCounts 方法中的逻辑用来处理步骤序列，而 flatMap 则用来处理 Try 容器。注意返回的是子类型，要么返回 Success，要么返回 Failure。我们在 17.2 节将会看到 scala.concurrent.Future（http://www.scala-lang.org/api/current/scala/concurrent/Future.html）也是 Monad 性质的。

Monad 首先在 Haskell 中使用[1]，其中 Haskell 十分强调函数的纯性。例如，Monad 在纯代码中被用于区分输入与输出（I/O）。IO Monad 能处理这种不同的关注点。另外，由于 Monad 出现在使用它的函数的签名中，读者和编译器都知道该函数不是纯函数。类似地，出于相同的目的，在很多语言中也定义了 Reader 和 WriterMonad。

Monad 的推广是 Arrow。Monad 将一个值提升为上下文，也就是说，传递给 flatMap 的函数的类型为 A => M[B]，而 Arrow 将函数提升为上下文，即（A => B）=> C [A => B]。Arrow 的组合可以表示一系列的处理步骤，也就是先执行 A => B，再执行 B => C 等。这种用法的引用是透明的，在实际的上下文环境之外。与此相反，传递给 flatMap 的函数明确知道它的上下文，这一点体现在返回值中！

## 16.3　本章回顾与下一章提要

我希望这篇对高级概念的简短介绍，足够帮助你理解人们常提起的一些概念。如果这些概念理解起来有困难，我希望本篇的介绍能够帮助你进一步理解这些概念如此强大的原因。

Scala 的标准库采用面向对象而不是用范畴的方法来增加函数，如 map、flatMap 和 unit。

注 1：Philip Wadler 在其个人主页（http://homepages.inf.ed.ac.uk/wadler/topics/monads.html）上写了多篇前沿性文章，探讨 Monad 理论及其应用。

然而，通过像 `flatMap` 一样的方法，我们可以获得“具有 Monad 属性”的行为，让 `for` 推导式更简洁。

我曾不经意地提到 Monad、Functor、Applicative 和 Arrow 等概念，用来作为函数式设计模式的例子。尽管这些概念对某些函数式编程开发者来说很难理解，但在面向对象编程中，无论采取什么形式，模式的过度使用并没有让可复用构造器的核心思想失效，

不幸的是，范畴一直神秘不可知。这是因为复杂的数学表达形式和名称使得大部分开发者很难理解它们。但它们本质上是对我们熟悉的概念的抽象，对程序的正确性、合理性、简洁性和表达力很有意义。我们希望这些概念在开发者中逐渐变得熟知起来。

附录 A 中列出了一些书籍、论文和博文，用于对函数式编程进行更深层次的探讨。有几点值得在这里强调。你可能会研究到的其他两个函数式结构有 Lense 和 Monad Transforme。前者用于获取或设置（设置会引起实例复制）嵌套在实例中的值，后者用于将 Monad 组合起来。

Paul Chiusano 与 Rúnar Bjarnason 的《Scala 函数式编程》是你进一步学习函数式编程知识的有用资料，书中还提供了大量的 Scala 练习。该书的作者是 Scalaz 的主要贡献者。Eugene Yokota 在博客上连续发表了多篇关于 Scalaz 学习的相当棒的文章。

Shapeless 网站对高级的构造技巧，尤其是类型系统中的构造进行了探索。聚合类网站 http://typelevel.org 里面也有不少具有启发性的项目。

下一章主要介绍一些更实用的技能，即如何运用 Scala 编写高并发软件。

# 第 17 章 并发工具

21 世纪初期，摩尔定律已经不大适用单核 CPU，*多核问题*（multicore problem）不断得到人们的关注。通过增加 CPU 的核数和服务器数量，并使用水平扩展取代垂直扩展，我们能够继续对性能进行扩展。

水平扩展要求开发人员编写并发软件，这为开发人员带来了挑战。并发并不容易处理，这类应用需要对共享可变状态的访问进行协调，这意味需要处理使用锁、互斥量及信号量这样的工具的多线程编程。如果未能正确地协调这些访问，便会产生像第 2 章中提到的那种不可预见的行为。在第 2 章中，其他线程突然对你所使用的一些变量进行了修改，这也意味着代码中存在竞态条件和锁竞争。

当人们意识到利用*不可变性*（immutability）和纯函数化能够解决这些问题时，函数式编程开始变得主流起来。我们也能看到 actor 模型这样的古老并发方法重新变得充满活力。

本章将对 Scala 中的并发工具进行深入讲解。当然，你也可以使用曾在 Java 中运用的任意并发机制（包括多线程 API、消息队列等）。但本章中我们只会讨论 Scala 的专有工具，首先讲解的 API 适用于一个非常古老的场景：多个协同工作的单线程进程。

## 17.1 scala.sys.process包

某些场景下，我们可以使用小的、同步的进程通过数据库事务、消息队列或进程间的数据转移来完成同步状态。

`scala.sys.process` 包（http://www.scala-lang.org/api/current/scala/sys/process/package.html）提供了一套 DSL 方言，可用于运行或管理操作系统进程，同时也能对进程 I/O 进行处理。下面我们将通过一个 REPL 会话对这套 DSL 方言提供的一些功能进行演示。请注意，我们需要在 `bash` 这样的 Unix shell 环境下执行这些命令：

```
// src/main/scala/progscala2/concurrency/process/processes.sc
scala> import scala.sys.process._
scala> import scala.language.postfixOps
scala> import java.net.URL
scala> import java.io.File

// 执行命令,并写入标准输出。
scala> "ls -l src".!
total 0
drwxr-xr-x  4 deanwampler  staff  136 Dec 19  2013 main
drwxr-xr-x  4 deanwampler  staff  136 Dec 19  2013 test
res33: Int = 0

// 将命令相关标记传入Seq对象,执行命令后将返回一个记录了命令输出信息的字符串。
scala> Seq("ls", "-l", "src").!!
res34: String =
"total 0
drwxr-xr-x  4 deanwampler  staff  136 Dec 19  2013 main
drwxr-xr-x  4 deanwampler  staff  136 Dec 19  2013 test
"
```

我们还能将多个进程串联起来，如下所示：

```
// 创建一个进程,用于访问URL资源,该进程的输出将重新定向到"grep $filter"命令的输入中,
   与此同时,grep操作的输出将会以追加的方式(非覆写)输入到文件中。
def findURL(url: String, filter: String) =
  new URL(url) #> s"grep $filter" #>> new File(s"$filter.txt")

// 对输出文件执行ls -l命令。假如该文件存在,则计算文件行数。
def countLines(fileName: String) = s"ls -l $fileName" #&& s"wc -l $fileName"
```

我们可以使用 DSL 中定义的 `#>` 方法对文件进行覆写，也可以使用该方法使用管道将输出定向到另一个程序的标准输入中。`#>>` 方法只能用于覆写文件。只有当 `#&&` 方法左侧的进程成功结束时，该方法才会执行它右侧的进程。这也意味着左侧进程的结束代码（`Exit Code`）为 0。调用 `#>>` 方法和 `#&&` 方法都会返回 `scala.sys.process.ProcessBuilder`（http://www.scala-lang.org/api/current/#scala.sys.process.ProcessBuilder）对象。这两个方法都不会执行操作系统命令。我们需要调用 `ProcessBuilder` 对象的 ! 方法来执行命令：

```
scala> findURL("http://scala-lang.org", "scala") !
res0: Int = 0

scala> countLines("scala.txt") !
-rw-r--r--  1 deanwampler  staff  4111 Jul 31 22:35 scala.txt
      43 scala.txt
res1: Int = 0

scala> findURL("http://scala-lang.org", "scala") !
res2: Int = 0

scala> countLines("scala.txt") !
-rw-r--r--  1 deanwampler  staff  8222 Jul 31 22:35 scala.txt
      86 scala.txt
res3: Int = 0
```

由于每次执行时我们都会在文件中添加文本，因此文件行数变成了之前的两倍。

如果这类小的同步进程能够满足我们的设计需求，我们也可以在 Scala 或其他语言中实现这些进程，之后再调用 `process` 包中定义的 API 将这些进程粘合起来。

## 17.2 Future类型

对于某些需求而言，使用多进程实现并发显得太粗粒度了。我们需要在单一进程内简单地使用并发原语来实现并发。也就是说，我们需要一个比传统多线程 API 更高层次的 API，这个 API 更强调合理直观地构建代码块。

假设你希望以异步的方式运行几项工作，这些工作就不会阻塞彼此运行。比如说，这些工作也许会执行一些 I/O 操作。那么针对这类情况，使用 `scala.concurrent.Future`（http://www.scala-lang.org/api/current/scala/concurrent/Future.html）类便是最简单的方案。

一旦完成了 `Future` 对象的构建工作，控制权便会立刻返还给调用者，但结果值却无法保证立刻可用。`Future` 实例是一个句柄，它指向最终可用的结果值。无论操作成功与否，在 future 操作执行完毕之前，你可以继续执行其他工作。Scala 提供了多种方法用于处理 future 操作的执行。[1]

在 2.5.3 一节中，我们曾通过示例对隐式参数进行了讲解，该示例利用 `scala.concurrent.ExecutionContext`（http://www.scala-lang.org/api/current/#scala.concurrent.ExecutionContext）管理并运行 Future 对象。我们还在示例中应用了 `ExecutionContext.global`（http://www.scala-lang.org/api/current/#scala.concurrent.ExecutionContext$）对象，其中 `global` 对象利用 `java.util.concurrent.ForkJoinPool`（http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166-4jdk7docs/java/util/concurrent/ForkJoinPool.html）对象对线程池进行管理，而 `ForkJoinPool` 对象也会执行那些封装在 `Future` 对象中的任务。作为 Scala 的用户，我们并不需要关心 Scala 会如何运行我们的同步代码，除非是那些性能调优的特殊场景。（`ForkJoinPool` 是 JDK7 类库的一部分，而由于目前 Scala 还支持 JDK6，因此 Scala 将 Doug Lea 所实现的 `ForkJoinPool` 移植到了类库中，该实现之后也被 JDK7 录入。）

为了能够对 `Future` 类型有更深入的理解，我们先考虑这样一个场景：我们需要并行执行十件事情，之后将执行结果合并。

```
// src/main/scala/progscala2/concurrency/futures/future-fold.sc
import scala.concurrent.{Await, Future}
import scala.concurrent.duration.Duration
import scala.concurrent.ExecutionContext.Implicits.global

val futures = (0 to 9) map {                                    // ❶
  i => Future {
    val s = i.toString                                          // ❷
    print(s)
```

注 1：在某些场合下，`Promise`（http://www.scala-lang.org/api/current/#scala.concurrent.Promise）类还可用于与 `Future` 对象协同工作。具体请参考 Scala 的相关文档（http://docs.scala-lang.org/overviews/core/futures.html）。

```
      s
    }
  }

  val f = Future.reduce(futures)((s1, s2) => s1 + s2)            // ❸

  val n = Await.result(f, Duration.Inf)                          // ❹
```

❶ 创建了 10 个 Future 对象，每个对象都会执行一些操作。

❷ Future.apply 方法带有两个参数列表。第一个参数列表中只包含一个需要并发执行内容的*命名方法体*（by-name body），而第二个参数列表包含了隐式的 ExecutionContext 对象。我们将使用这个全局的隐式值。输入的方法体会把整数转化为字符串，打印并返回该字符串。future 对象的类型为 IndexedSeq[Future[String]]。在我们设计的这个示例中，Future 对象会立刻执行完毕。

❸ 把一组 Future 类型压缩成一个单独的 Future[String] 对象。在本示例中，该过程会把每个 Future 对象返回的字符串合并为一个字符串。

❹ 直到 Future f 完成之前，我们通过 scala.concurrent.Await 对象阻塞代码。输入的 Duration（http://www.scala-lang.org/api/current/#scala.concurrent.duration.Duration）参数则表示，如果需要的话，代码便会一直等待下去。如果你需要等待 Future 对象完成，那么选择 Await 对象是阻塞当前线程的较好方法。

需要强调至关重要的一点，执行 Future 体中的 print 语句时，print 语句的输出是无序的。例如：两次执行脚本时分别输出了 0214679538 和 0123467985。不过，由于 fold 方法会依照 Future 对象构造的顺序遍历这些对象，因此 fold 方法生成的字符串总是会严格按数值次序排列，即 0123456789。

Future.fold 方法以及其他相似的方法（http://www.scala-lang.org/api/current/#scala.concurrent.Future$）本身也是异步执行的，这些方法将返回一个新的 Future 对象。在我们的示例中，只有调用 Await.result 方法时，程序才会阻塞。

通常，我们并不希望等待执行结果时阻塞程序。我们只希望当 Future 执行结束后，系统会执行少量的代码。而注册回调方法能帮助我们实现这一功能。举个例子，简单的网络服务器会创建 Future 对象用于对请求进行处理，同时利用回调将执行的结果返还给调用者。下面的示例对相关的逻辑进行了解释：

```
// src/main/scala/progscala2/concurrency/futures/future-callbacks.sc
import scala.concurrent.Future
import scala.concurrent.duration.Duration
import scala.concurrent.ExecutionContext.Implicits.global

case class ThatsOdd(i: Int) extends RuntimeException(            // ❶
  s"odd $i received!")

import scala.util.{Try, Success, Failure}                        // ❷

val doComplete: PartialFunction[Try[String],Unit] = {            // ❸
  case s @ Success(_) => println(s)                              // ❹
  case f @ Failure(_) => println(f)
```

```
}

val futures = (0 to 9) map {                                  // ❺
  case i if i % 2 == 0 => Future.successful(i.toString)
  case i => Future.failed(ThatsOdd(i))
}
futures map (_ onComplete doComplete)                         // ❻
```

❶ 如果发现奇数，我们将抛出异常。

❷ 导入 scala.util.Try（http://www.scala-lang.org/api/current/#scala.util.Try）类及其子类：Success 类和 Failure 类。

❸ 无论执行结果是否成功，我们为这两个结果定义相同的回调处理程序。为了对成功和失败事件进行封装，封装回调函数的输入参数是 Try[A] 类型，因此回调函数的类型是 PartialFunction[Try[String],Unit]，其中 A 是 String 类型。由于回调函数异步执行，不会返回任何事物，因此回调函数的返回类型是 Unit 类型。如果我们需要构建 web 服务器，回调函数应该向调用者发送回复信息。

❹ 如果 Future 任务执行成功，执行结果便能与 Success 子句匹配，否则结果将与 Failure 匹配。无论最终结果如何，我们都将打印执行结果。

❺ 创建一组 Future 对象，假如出现奇数，这些 Failure 将会报错。为了能够立刻返回 Success 或 Failure 对象，我们使用了 Future 伴生对象的两个方法。

❻ 遍历 future 对象，为每个对象附上回调方法，一旦我们的 Future 对象执行完毕，便会触发这些回调方法。

执行这段脚本将产生下列输出，而每次运行时输出内容的顺序各不相同：

```
Success(0)
Success(2)
Failure($line137.$read$$iw$$iw$ThatsOdd: odd 1 received!)       // ❶
Success(4)
Failure($line137.$read$$iw$$iw$ThatsOdd: odd 3 received!)
Success(6)
Success(8)
Failure($line137.$read$$iw$$iw$ThatsOdd: odd 5 received!)
Failure($line137.$read$$iw$$iw$ThatsOdd: odd 9 received!)
Failure($line137.$read$$iw$$iw$ThatsOdd: odd 7 received!)
```

❶ 编译脚本中定义的 ThatsOdd 对象时，编译器会为该对象合成一个并非很优雅的名字。

在下一节中，我们会看到更多应用了 Future 类型的示例。

与 Option 类型、Try 类型、Either 类型以及其他的容器类型相似，Future 类型也是“一元”的。我们可以在 for 推导式中使用这些类型，并使用像 map、flatMap、filter 这样的组合器对执行结果进行处理。

## Async类

使用 Future 类时，我们需要大量使用回调，但这使得代码很快变得复杂起来。因此处理一组有关联的任务时，我们可以把一些 Future 对象组合起来以减少回调数量。Scala 新

设计的 scala.async.Async 模块可以使用户更容易地构建这类计算。SIP-22（SIP 是 Scala Improvement Process 的简写，意为 Scala 改进流程，SIP-22 的网址为 http://docs.scala-lang.org/sips/pending/async.html）对该模块进行了描述。Scala 2.10 和 Scala 2.11 均在 Github（https://github.com/scala/async）中实现了这一功能，并将该模块作为“可选模块”进行分发（请参考表 21-11，了解 Scala 2.11 版本分发的可选模块）。

Async 模块中提供了两个基本方法，可用于同步代码块中：

```
def async[T](body: => T): Future[T]            // ❶
def await[T](future: Future[T]): T             // ❷
```

❶ 启动异步计算，并立刻返回对应的 Future 对象。

❷ 等待 Future 执行完毕。

在下面的例子中，我们模拟了一组同步调用，在第一次同步调用中，我们首先判断指定 id 的“记录”是否存在。假如该记录存在，我们将返回记录；否则，便返回错误记录。

```
// src/main/scala/progscala2/concurrency/async/async.sc
import scala.concurrent.{Await, Future}
import scala.concurrent.duration.Duration
import scala.async.Async.{async, await}
import scala.concurrent.ExecutionContext.Implicits.global

object AsyncExample {
  def recordExists(id: Long): Boolean = {                          // ❶
    println(s"recordExists($id)...")
    Thread.sleep(1)
    id > 0
  }

  def getRecord(id: Long): (Long, String) = {                      // ❷
    println(s"getRecord($id)...")
    Thread.sleep(1)
    (id, s"record: $id")
  }

  def asyncGetRecord(id: Long): Future[(Long, String)] = async {    // ❸
    val exists = async { val b = recordExists(id); println(b); b }
    if (await(exists)) await(async { val r = getRecord(id); println(r); r })
    else (id, "Record not found!")
  }
}

(-1 to 1) foreach { id =>                                          // ❹
  val fut = AsyncExample.asyncGetRecord(id)
  println(Await.result(fut, Duration.Inf))
}
```

❶ 此处声明了一个执行时间较长的谓词（predicate），用于测试记录是否存在。假如输入的 id 大于 0，那么该谓词将返回 true。

❷ 另一个执行时间较长的方法，该方法会根据输入的 id 值返回对应的记录。

❸ asyncGetRecord 方法将多个异步操作组合起来顺序执行。该方法首先将以异步方式调用

recordExists 方法。之后，asyncGetRecord 方法会等待 recordExists 方法的执行结果。如果结果为 true，asyncGetRecord 方法便会以异步的方式读取相关记录；反之，则会返回错误记录。

❹ 使用三个下标调用 asyncGetRecord 方法。

执行该脚本将产生下列结果（运行几秒后才会产生结果）：

```
recordExists(-1)...
false
(-1,Record not found!)
recordExists(0)...
false
(0,Record not found!)
recordExists(1)...
true
getRecord(1)...
(1,record: 1)
(1,record: 1)
```

请注意，只有输入“合理”的下标 1 时，getRecord 方法才会被调用一次。

与串行化 Future 对象相比，利用 Async 模块编写出的代码更为整洁；尽管仍然没有真正的同步代码那么易于理解，不过你能够从异步执行中获益。

无论是否使用了 Async，运用 Future 类型仅仅是解决并发的一种手段，它并不能称为全局的策略。Future 并没有为用户在应用程序的层面上提供管理分布式进程的大量工具，包括错误处理。而 actor 模型则满足这些需求！

## 17.3 利用Actor模型构造稳固且可扩展的并发应用

Actor 最初是为了进行人工智能研究而设计的。Carl Hewiit 和他的合作者在 1973 年的一篇论文（访问 arixiv.org 可以查阅到 2014 年的修改版，http://arxiv.org/pdf/1008.1459.pdf）中对其进行了描述，而 Gual Agha 也曾在 1973 年出版的 *Actors*（MIT 出版）一书中描述了 actor 模型的相关理论。Erlang 语言及其虚拟机将 Actor 模型视为核心概念。在 Scala 这样的其他语言中，actor 模型则以类库的方式实现，且它与其他的并发抽象思想一并实现。

actor 本质上是一个可以接收并处理消息的对象，它一次接收一个消息，并且不允许消息抢占。在某些 actor 系统中，消息到来的顺序并不重要，但并不是所有的 actor 系统都是这样设计的。actor 对象也许会在对象内部对消息进行处理，也可能会转发消息，还有可能向其他 actor 对象发送消息。在处理消息时，某些 actor 还会创建新的 actor 对象。假如我们使用 actor 模型实现状态机，当状态转化时，某些 actor 也许会因此修改消息处理逻辑。

传统的对象系统通过方法调用传递消息。与这些系统不同，actor 通常以异步的方式发送消息，这导致 actor 执行操作的全局顺序是不确定的。与传统的对象相似，actor 在处理消息时也许也会对状态进行控制。设计良好的 actor 系统即使无法完全防止其他代码直接访问和修改状态，至少也会努力阻止这种行为。

正是因为 actor 系统具有这些特性，即便是在跨集群的环境下，actor 也能并发执行。actor 系统提供了用于管理全局状态的规则方法，这能在很大程度上避免（并非完全避免）传统多线程并发存在的问题。

## 17.4　Akka：为Scala设计的Actor系统

2009 年，本书第 1 版的编写工作完成时，Scala 自带了一套 actor 库。在第 1 版中，我们也为这套库编写了示例。不过，从那时以后，一个全新的 actor 系统就此启动。该系统名为 Akka 库（http://akka.io），它完全不依赖于之前的 actor 库。

现在，Scala 已经将原先的 actor 库从类库中移除，而将 Akka 视为处理基于 actor 并发模型的官方类库。Akka 是一个独立项目。Scala 和 Akka 都由 Typesafe（http://typesafe.com）开发并提供支持的，同时 Akka 也提供了一套完整的 Java API。

在 1.4 节中，我们运用 Akka 编写过一个简单的并发示例。下面我们将实现一个更为真实的示例。随着学习的深入，你将发现 Akka Scaladoc（http://doc.akka.io/api/akka/current/）越来越有用。

在 actor 模型的所有实现中，Erlang 和 Akka 是最重要的两个实现，也是在产业界得到最广泛应用的实现。Akka 的灵感来源于 Erlang 的 actor 模型实现。这两个实现都是重要的创新，都实现了一个兼具错误处理和恢复原状功能的健壮模型。

并不是只有 actor 可以执行系统定义的正常工作，你也可以创建*监督者*（supervisor）对象监控一个或多个 actor 的生命周期。假如某个 actor 运行时抛出异常而导致执行失败，监督者便会按照某一策略恢复原状。监督者所遵循的策略包含重启策略、关闭策略、应该忽视错误还是将错误交由相关监督者处理策略。

重启 actor 时，假如其他 actor 与出错的 actor 合作紧密，并且这些 actor 都接受相同的监督者管理，那么我们应选择 all-for-one 策略，重启所有的 actor。假如那些受管理的 actor 彼此之前没有关联，出错的 actor 不会对其他的 actor 造成影响，那么我们应该使用 one-for-one 策略。使用这种策略后，只有出错的 actor 需要重启。

这种架构很清晰地将错误处理逻辑从正常流程中剥离出来，进而为错误处理提供了架构级处理策略。最重要的一点，该架构对“任其崩溃”（let  it crash）的观点进行了提升。

我们通常都会把错误处理代码和正常处理代码混在一起，这就会导致代码复杂且混乱，不易实现一套完整全面的处理策略。在某些实际的生产环境中不可避免地会出现一些失败的故障恢复，这将使整个系统处于不一致的状态。假如程序出现了无法避免的崩溃，服务只能对此进行一些妥协，但是诊断实际的问题源也是很困难的。

我们接下来将进行示例讲解，该示例将对客户端接口进行模拟，其调用的服务会将工作分派给工作线程。这个客户端接口（我们同时也在客户端接口中定义 `main` 方法）取名为 `AkkaClient`。`AkkaClient` 会将用户命令传递给一个 `ServerActor` 对象，而该对象则会把这些工作转发给许多个 `WorkerActor` 对象，因此 `ServerActor` 对象并不会被这些工作堵塞。每个工作线程都模拟了一个具有分片功能的数据存储单元。该存储单元维护了一个持有 key（`Long` 类型）和 value（字符串）的 `map` 对象，也支持了 CRUB（CRUB 是 create、read、

update 和 delete 的简写，分别代表了创建、读取、更新以及删除操作）的语义。

AkkaClient 可以构造出 akka.actor.ActorSystem（http://doc.akka.io/api/akka/current/#akka.actor.ActorSystem）对象，该对象会对整个 actor 系统进行控制。在任何一个应用中，我们都会遇到一个或若干 akka.actor.ActorSystem 对象。AkkaClient 之后构造出一个 ServerActor 实例，并向该实例发送一条表示启动系统的消息。最后，AkkaClient 提供了一个简单的命令行接口供用户使用。

在学习 AkkaClient 之前，我们先对 Message 对象进行了解，该对象中定义了在 actor 之间交换的所有消息体：

```
// src/main/scala/progscala2/concurrency/akka/Messages.scala
package progscala2.concurrency.akka
import scala.util.Try

object Messages {                                                // ❶
  sealed trait Request {                                         // ❷
    val key: Long
  }
  case class  Create(key: Long, value: String) extends Request   // ❸
  case class  Read(key: Long) extends Request                    // ❹
  case class  Update(key: Long, value: String) extends Request   // ❺
  case class  Delete(key: Long) extends Request                  // ❻

  case class  Response(result: Try[String])                      // ❼

  case class  Start(numberOfWorkers: Int = 1)                    // ❽
  case class  Crash(whichOne: Int)                               // ❾
  case class  Dump(whichOne: Int)                                // ❿
  case object DumpAll
}
```

❶ Messages 对象中包含了所有的消息类型定义。

❷ Request 是所有 CRUB 请求的父特征，所有的请求都使用一个 Long 类型的 key 值。

❸ 构造一个新的“记录”，并指定该记录的 key 和 value 值。

❹ 读取指定 key 值的记录。

❺ 对指定 key 值的记录进行更新（假如记录不存在，创建一条新记录），为其指定新的 value 值。

❻ 删除指定 key 值的记录，假如不存在满足条件的 key 值，则不执行任何操作。

❼ 将回复信息封装到一条普通的消息体中。我们使用 scala.util.Try 类型对结果值进行封装，封装后的对象能够指明操作成功还是失败。

❽ 启动系统。这条消息将被发送给 ServerActor 对象，该消息中包含了应创建多少个工作节点的信息。

❾ 向某一工作节点发送此消息，以模拟“崩溃”事件。

❿ 发送这条消息，以便“获取”某一工作节点或全部工作节点的状态信息。

现在，我们将学习 AkkaClient 对象的具体实现：

```
// src/main/scala/progscala2/concurrency/akka/AkkaClient.scala
package progscala2.concurrency.akka
import akka.actor.{ActorRef, ActorSystem, Props}
import java.lang.{NumberFormatException => NFE}

object AkkaClient {                                                // ❶
  import Messages._

  private var system: Option[ActorSystem] = None                   // ❷

  def main(args: Array[String]) = {                                // ❸
    processArgs(args)
    val sys = ActorSystem("AkkaClient")                            // ❹
    system = Some(sys)
    val server = ServerActor.make(sys)                             // ❺
    val numberOfWorkers =                                          // ❻
      sys.settings.config.getInt("server.number-workers")
    server ! Start(numberOfWorkers)                                // ❼
    processInput(server)                                           // ❽
  }

  private def processArgs(args: Seq[String]): Unit = args match {
    case Nil =>
    case ("-h" | "--help") +: tail => exit(help, 0)
    case head +: tail => exit(s"Unknown input $head!\n"+help, 1)
  }
  ...
```

❶ AkkaClient 是一个对象，我们可以在它的作用域内定义 main 方法。

❷ 代码中只有一个 ActorSystem（http://doc.akka.io/api/akka/current/#akka.actor.ActorSystem）对象，我们将其保存在一个 Option 对象中。在关闭系统时，我们会使用 ActorSystem 对象，对此我们稍后进行讲解。请注意，system 是私有可变变量。不过由于 actor 会对并发行为进行控制，因此我们无需担心 system 变量的并发访问。

❸ main 方法首先对命令行参数进行处理。processArgs 当前实际上只支持一个参数选项 help 选项。

❹ 创建 ActorSystem 对象，并更新 system 的 Option 对象。

❺ 通过调用 ServerActor 伴生对象的 make 方法，构造出 ServerActor 的一个实例。

❻ 根据配置，决定使用多少工作节点。

❼ 向 ServerActor 对象发送 Start 消息体，启动系统。

❽ 对用户输入的命令行信息进行处理。

Akka 使用了 Typesafe 提供的 Config 库（https://github.com/typesafehub/config），对在文件中或通过程序定义的配置值进行管理。在本示例中，我们使用了下面的配置文件：

```
// src/main/resources/application.conf
akka {                                                             // ❶
  loggers  = [akka.event.slf4j.Slf4jLogger]                        // ❷
  loglevel = debug
```

```
    actor {                                                    // ❸
      debug {                                                  // ❹
        unhandled = on
        lifecycle = on
      }
    }
  }

  server {                                                     // ❺
    number-workers = 5
  }
```

❶ 对 Akka 系统的属性进行全局配置。

❷ 配置当前使用的日志模块。SBT 包含了 `akka-slf4j` 模块，该模块支持这个接口。与本配置文件同级的目录中存在一个对应的 `logback.xml` 文件，该文件对日志进行了配置（我们并未列出这些配置）。默认情况下，所有的 debug 和更高级别的消息都会被记录下来。

❸ 为每个 actor 对象进行属性配置。

❹ 如果某一 actor 收到了它无法处理的消息或生命周期事件，debug 级别的日志将会记录该事件。

❺ 由于 `ServerActor` 实例将会被命名为 `server`，因此该配置块会对 `ServerActor` 实例进行设置。该配置块中包含了一个自定义设置，用于决定使用工作节点的数量。

我们再回到 `AkkaClient` 的实现代码，`AkkaClient` 通过下面的这个长函数对用户输入进行处理：

```
  ...
  private def processInput(server: ActorRef): Unit = {          // ❶
    val blankRE = """^\s*#?\s*$""".r
    val badCrashRE = """^\s*[Cc][Rr][Aa][Ss][Hh]\s*$""".r
    val crashRE = """^\s*[Cc][Rr][Aa][Ss][Hh]\s+(\d+)\s*$""".r
    val dumpRE = """^\s*[Dd][Uu][Mm][Pp](\s+\d+)?\s*$""".r
    val charNumberRE = """^\s*(\w)\s+(\d+)\s*$""".r
    val charNumberStringRE = """^\s*(\w)\s+(\d+)\s+(.*)$""".r

    def prompt() = print(">> ")                                 // ❷
    def missingActorNumber() =
      println("Crash command requires an actor number.")
    def invalidInput(s: String) =
      println(s"Unrecognized command: $s")
    def invalidCommand(c: String): Unit =
      println(s"Expected 'c', 'r', 'u', or 'd'. Got $c")
    def invalidNumber(s: String): Unit =
      println(s"Expected a number. Got $s")
    def expectedString(): Unit =
      println("Expected a string after the command and number")
    def unexpectedString(c: String, n: Int): Unit =
      println(s"Extra arguments after command and number '$c $n'")
    def finished(): Nothing = exit("Goodbye!", 0)
```

```
  val handleLine: PartialFunction[String,Unit] = {              // ❸
    case blankRE() =>   /* do nothing */
    case "h" | "help" => println(help)
    case dumpRE(n) =>                                           // ❹
      server ! (if (n == null) DumpAll else Dump(n.trim.toInt))
    case badCrashRE() => missingActorNumber()                   // ❺
    case crashRE(n) => server ! Crash(n.toInt)
    case charNumberStringRE(c, n, s) => c match {               // ❻
      case "c" | "C" => server ! Create(n.toInt, s)
      case "u" | "U" => server ! Update(n.toInt, s)
      case "r" | "R" => unexpectedString(c, n.toInt)
      case "d" | "D" => unexpectedString(c, n.toInt)
      case _ => invalidCommand(c)
    }
    case charNumberRE(c, n) => c match {                        // ❼
      case "r" | "R" => server ! Read(n.toInt)
      case "d" | "D" => server ! Delete(n.toInt)
      case "c" | "C" => expectedString
      case "u" | "U" => expectedString
      case _ => invalidCommand(c)
    }
    case "q" | "quit" | "exit" => finished()                    // ❽
    case string => invalidInput(string)                         // ❾
  }

  while (true) {
    prompt()                                                    // ❿
    Console.in.readLine() match {
      case null => finished()
      case line => handleLine(line)
    }
  }
}
...
```

❶ 定义了一些正则表达式，用于解析输入信息。

❷ 定义了一些嵌套方法，分别用于打印提示符、汇报错误、结束处理和关闭系统。

❸ `handleLine` 是主要的处理方法，考虑到偏函数语法的便利性，我们将其定义为偏函数。`handleLine` 函数首先会对空行进行匹配（同时也会匹配"注释行"，即第一个非空白字符是 `#` 字符的输入行），将空行过滤出去。之后便会处理用户的帮助请求（匹配 `h` 或 `help` 字符串）。

❹ 要求一个或全部工作节点输出其状态信息，状态信息中包含存储了一组 key 值对的"数据存储"信息。

❺ 为了能够解释 Akka 如何对 actor 进行管理，处理消息后，某一工作节点便会崩溃。接受到该输入时，我们首先检查用户是否忘记指定 actor 对应的数字。假如语法正确，再对正确的输入进行处理，并将其发送给 `ServerActor` 对象。

❻ 假如命令中同时包含字母、数值和字符串，该命令一定是一条"创建"或"更新"的命令。如果命令类型匹配的话，`handleLine` 方法会将该命令发送给 `ServerActor` 对象。如果不匹配，则会记录错误。

❼ 与之相似，假如用户输入中只包含命令字符和数字，该输入一定是一条“读取”或“删除”的命令。

❽ 提供了三种退出应用的方式（输入 Ctrl-D 也能退出应用）。

❾ 假如输入不满足之前的正则表达式，便输入一个错误。

❿ 打印初始化提示符，之后循环等待用户输入并对每行输入进行处理。

请注意，假如用户输入了无效的用户命令，系统并不会崩溃。由于我们并没使用像 Gnu readline 这样的辅助库，因此很不幸程序对回退键的处理并不正确。

最后，我们再输入下列信息，完成该代码文件：

```
  ...
  private val help =                                          // ❶
  """Usage: AkkaClient [-h | --help]
    |Then, enter one of the following commands, one per line:
    |  h | help      Print this help message.
    |  c n string    Create "record" for key n for value string.
    |  r n           Read record for key n. It's an error if n isn't found.
    |  u n string    Update (or create) record for key n for value string.
    |  d n           Delete record for key n. It's an error if n isn't found.
    |  crash n       "Crash" worker n (to test recovery).
    |  dump [n]      Dump the state of all workers (default) or worker n.
    |  ^d | quit     Quit.
    |""".stripMargin

  private def exit(message: String, status: Int): Nothing = {        // ❷
    for (sys <- system) sys.shutdown()
    println(message)
    sys.exit(status)
  }
}
```

❶ 详细的帮助消息。

❷ 用于辅助系统退出的辅助函数。假如 ActorSystem 已开启，此函数会关闭该系统。之后打印一条信息并退出程序。

接下来，我们看一下 ServerActor 的实现：

```
// src/main/scala/progscala2/concurrency/akka/ServerActor.scala
package progscala2.concurrency.akka
import scala.util.{Try, Success, Failure}
import scala.util.control.NonFatal
import scala.concurrent.duration._
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import akka.actor.{Actor, ActorLogging, ActorRef,
  ActorSystem, Props, OneForOneStrategy, SupervisorStrategy}
import akka.pattern.ask
import akka.util.Timeout


class ServerActor extends Actor with ActorLogging {              // ❶
```

```
import Messages._

implicit val timeout = Timeout(1.seconds)

override val supervisorStrategy: SupervisorStrategy = {          // ❷
  val decider: SupervisorStrategy.Decider = {
    case WorkerActor.CrashException => SupervisorStrategy.Restart
    case NonFatal(ex) => SupervisorStrategy.Resume
  }
  OneForOneStrategy()(decider orElse super.supervisorStrategy.decider)
}

var workers = Vector.empty[ActorRef]                             // ❸

def receive = initial                                            // ❹

val initial: Receive = {                                         // ❺
  case Start(numberOfWorkers) =>
    workers = ((1 to numberOfWorkers) map makeWorker).toVector
    context become processRequests                               // ❻
}

val processRequests: Receive = {                                 // ❼
  case c @ Crash(n) => workers(n % workers.size) ! c
  case DumpAll =>                                                // ❽
    Future.fold(workers map (_ ? DumpAll))(Vector.empty[Any])(_ :+ _)
      .onComplete(askHandler("State of the workers"))
  case Dump(n) =>
    (workers(n % workers.size) ? DumpAll).map(Vector(_))
      .onComplete(askHandler(s"State of worker $n"))
  case request: Request =>
    val key = request.key.toInt
    val index = key % workers.size
    workers(index) ! request
  case Response(Success(message)) => printResult(message)
  case Response(Failure(ex)) => printResult(s"ERROR! $ex")
}

def askHandler(prefix: String): PartialFunction[Try[Any],Unit] = {
  case Success(suc) => suc match {
    case vect: Vector[_] =>
      printResult(s"$prefix:\n")
      vect foreach {
        case Response(Success(message)) =>
          printResult(s"$message")
        case Response(Failure(ex)) =>
          printResult(s"ERROR! Success received wrapping $ex")
      }
    case _ => printResult(s"BUG! Expected a vector, got $suc")
  }
  case Failure(ex) => printResult(s"ERROR! $ex")
}

protected def printResult(message: String) = {
  println(s"<< $message")
```

```
  }

  protected def makeWorker(i: Int) =
    context.actorOf(Props[WorkerActor], s"worker-$i")
}

object ServerActor {                                              // ❾
  def make(system: ActorSystem): ActorRef =
    system.actorOf(Props[ServerActor], "server")
}
```

❶ 混入 `ActorLogging`（http://doc.akka.io/api/akka/current/#akka.actor.ActorLogging）特征，该 trait 会增加用于记录信息的 `log` 字段。

❷ 使用 `akka.actor.SupervisorStrategy`（http://doc.akka.io/api/akka/current/#akka.actor.SupervisorStrategy）覆写默认的监管策略。假如发生了我们模拟的崩溃事件，`ServerActor` 便会重启。假如出现了 `NonFatal`（http://www.scala-lang.org/api/current/#scala.util.control.NonFatal$）异常，系统将继续运行（这样做会有风险！）。由于这些工作节点相互独立，因此我们选用了 one-for-one 策略。假如该策略的 `Decider` 对象出错了，父类的监控器（supervisor）会代理执行监控策略。

❸ 使用 `akka.actor.ActorRef`（http://doc.akka.io/api/akka/current/#akka.actor.ActorRef）实例追踪执行工作的 actor 对象，这些实例会引用 actor 对象。

❹ 定义 `recive` 方法，将其设置为 `initial` 请求处理器。

❺ 为了能够为开发人员提供方便，Actor（http://doc.akka.io/api/akka/current/#akka.actor.Actor）类型中定义了 `Receive` 这一类型成员，该成员是 `PartialFunction[Any,Unit]` 的别名。而本代码行对 `Receive` 方法进行了声明，用于处理最初发送给 actor 的 `Start` 消息。最初，actor 对象只希望能够接收到 `Receive` 消息。假如接收到了其他消息，这些 actor 对象会将消息保存在 actor 的邮箱（mailbox）中，直到我们为这些消息定义了处理机制。我们可以将 `Receive` 方法视为该 actor 对象所使用的状态机中的第一个状态的实现方法。

❻ 假如 `ServerActor` 对象接收到了 `Start` 消息，该对象便会构造工作节点，并将状态机的相关状态进行切换，在新的状态下，`processRequests` 方法将负责处理消息。

❼ 接收到 `Start` 消息后，`Receive` 代码块会对之后收到的消息体进行处理。最开始的几条 case 类分别与 `Crash`、`DumpAll` 和 `Dump` 消息相匹配。而 `request: Request` 子句则负责处理 CRUB 命令。最后，`Receive` 代码块将对工作节点返回的 `Response` 消息进行处理。值得注意的是，我们将使用工作节点的数量对请求取模，由此计算出需要分配的工作节点索引，这也是工作节点向量的有效索引。`ServerActor` 对象收到工作节点的返回消息后会将该消息打印出来。

❽ 收到 `DumpAll` 消息后，我们需要将其转发给所有的工作节点，并且收集所有这些节点的回复信息并将这些信息汇总格式化成一条结果信息。通过应用 ask 模式我们实现了这一功能。但是我们必须导入 `akka.pattern.ask` 对象（导入代码位于代码文件的顶部）以使用这一功能。我们没有使用 ! 命令，而是通过 ? 命令发送消息。? 命令会返回一个 `Future` 对象，而使用 ! 发送消息，将返回一个 `Unit`。尽管这两种消息类型都是异步的，

但是使用 ask 模式时，消息接收方返回的回复消息将会作为 Future 对象的完成事件被系统捕获。我们使用 Future.fold 方法将一组 Future 对象合并为一个单独的 Future 对象，并将其封装到一个 Vector 对象中。之后，为了能够对执行完毕的 Future 对象进行处理，我们使用 onComplete 方法注册了回调函数 askHandler。你也许已经注意到了，通过这些操作，这些嵌套类型也变得更为复杂了。

❾ 该伴生对象提供了构造 actor 的便利方法：make 方法。该方法会使用构造 actor 的必需习语创建该对象。（我们将会在稍后谈论这点。）

构造 actor 对象时会返回 Receive 方法（Receive 是 PartialFunction[Any, Unit] 的别名），这时 Actor.receive 方法才会被调用一次，之后即使接收到方法调用请求，该方法也不会被调用。每收到一条消息时，构造的 Receive 方法便会被调用一次。Receive 方法中的消息处理子句是可以修改的，我们可以调用 Actor.become 方法使用新的 Receive 方法处理所有的消息。如果需要的话，我们可以利用这一特性实现一套复杂的状态机。你可以混入 FSM（finite state machine 的简写，即有限状态机，http://doc.akka.io/api/akka/current/#akka.actor.FSM）特征，以减少实现状态机所需要编写的代码。FSM 特征提供了一套用于定义状态机的方便 DSL 语法。

ServerActor 会把所有工作节点的回复消息打印到控制台上。因为 AkkaClient 对象并不是 actor 对象，所以 ServerActor 无法将这些回复信息发送给 AkkaClient 对象。当 ServerActor 调用 Actor.sender 方法（返回消息的最初发送方对应的 ActorRef 对象）时，实际上返回了 ActorSystem.deadLetters（http://doc.akka.io/api/akka/current/#akka.actor.ActorSystem）变量。

system.actorOf(Props[ServerActor], "server") 语句用于构造 ServerActor 对象。该语句解决了两类设计难题。首先，由于 ActorRef 对象对 actor 实例进行了封装，因此我们无法简单地通过执行 new ServerActor 语句构造 ServerActor 对象。Akka 需要适当的对 actor 实例进行封装，以便完成其他的初始化步骤。

其次，Akka 中之所以存在 Props（http://doc.akka.io/api/akka/current/#akka.actor.Props$）这样的单例对象，主要是为了解决如何生成 JVM 字节码这样一个问题。为了能够在集群部署环境下将 Actor 实例分发到不同节点中，这些 Actor 实例需要能够被序列化。关于 Actor 实例序列化的具体内容，可以参考 Akka docs（http://doc.akka.io/docs/akka/current/scala/remoting.html）。假如我们在其他的实例中创建了该 actor 实例，Scala 编译器需要将创建 actor 实例的对象信息也包含在 actor 实例中。这意味着假如我们查看了某些 actor 实例序列化后的字节码，也许能在其中发现其中嵌入的其他类实例信息。假如创建 actor 的实例无法被序列化，该 actor 便无法被传输到其他节点去。更糟的是，包含了 actor 实例的状态也许会封装在 actor 中，这可能会导致远端节点的不一致行为。而单例对象 Props 则会有效地防止这类问题。

最后，我将列出了 WorkerActor 的实现代码：

```
// src/main/scala/progscala2/concurrency/akka/WorkerActor.scala
package progscala2.concurrency.akka
import scala.util.{Try, Success, Failure}
import akka.actor.{Actor, ActorLogging}
```

```
class WorkerActor extends Actor with ActorLogging {
  import Messages._

  private val datastore = collection.mutable.Map.empty[Long,String]  // ❶

  def receive = {
    case Create(key, value) =>                                        // ❷
      datastore += key -> value
      sender ! Response(Success(s"$key -> $value added"))
    case Read(key) =>                                                 // ❸
      sender ! Response(Try(s"${datastore(key)} found for key = $key"))
    case Update(key, value) =>                                        // ❹
      datastore += key -> value
      sender ! Response(Success(s"$key -> $value updated"))
    case Delete(key) =>                                               // ❺
      datastore -= key
      sender ! Response(Success(s"$key deleted"))
    case Crash(_) => throw WorkerActor.CrashException                 // ❻
    case DumpAll =>                                                   // ❼
      sender ! Response(Success(s"${self.path}: datastore = $datastore"))
  }
}

object WorkerActor {
  case object CrashException extends RuntimeException("Crash!")      // ❽
}
```

❶ datastore 变量存储了一个由 key 值对组成的可变 map。由于 Receive 处理方法是线程安全的（Akka 自身确保了这点），并且该变量的 WorkerActor 是私有状态，因此此处使用可变对象是没有风险的。不过由于共享可变状态这一行为是危险的，因此我们切忌通过消息将该 map 返回给消息调用者。

❷ 在 datastore 的 map 对象上添加一个新的 key 值对，并给消息发送方发送一条回复消息。

❸ 尝试读取指定 key 所对应的 value 值。假如指定 key 不存在，我们可以通过调用 Try 方法中封装的 datastore(key) 方法，自动捕获抛出的异常，并将其封装到 Failure 对象中。反之，Try（http://www.scala-lang.org/api/current/index.html#scala.util.Try）方法将返回一个 Success 对象，该对象中封装了找到的 value 值。

❹ 找到对应的 key 值，并对相关 value 进行更新（假如未能找到对应的 key 值，则直接创建一个新的 key 值对）。

❺ 删除一条 key 值对。假如 key 并不存在，则不执行任何操作。

❻ 抛出 CrashException 异常，通过这种方式使 actor 对象“崩溃”。我们之前已经对 WorkerActor 的监督策略进行了配置。根据配置，一旦抛出异常，系统便会重启该 actor。

❼ 根据 datastoremap 变量的当前内容构造出字符串，并将该字符串作为 actor 的状态回复给发送方。

❽ 我们使用特殊的 CrashExcepion 消息模拟 actor 崩溃事件。

接下来，我们将在 sbt 提示符中运行该示例：

```
run-main progscala2.concurrency.akka.AkkaClient
```

（你也可以调用 run 命令，并从显示的列表选择想要运行的编号。）输入 h 选项便能查看命令列表，而且可以尝试运行多种命令。输入 quit 则会退出 sbt。另外，我们还可以在 shell 窗口或命令窗口中输入下列命令运行程序：

```
sbt "run-main progscala2.concurrency.akka.AkkaClient" < misc/run-akka-input.txt
```

由于这些操作天生就是异步操作，因此每次执行该脚本，都会打印出不同的结果。即使从 misc/run-akka-input.txt 文件中复制出一些输入行再执行脚本，每次执行的结果仍然不同。

值得注意的是，actor 崩溃，数据便会一同丢失。如果数据丢失情况是不可接受的，你可以使用 Akka 持久化模块（http://doc.akka.io/docs/akka/current/scala/persistence.html）来恢复数据。持久化模块能够持久地存储 actor 的状态信息，因此即使重启 actor，它也能够恢复之前的状态信息。

你也许会担心，如果某个 actor 对象崩溃了，ServerActor 对象所持有的一组工作节点会不会变得无效。这解释了所有对 actor 的访问都必须通过一个中间“手柄”ActorRef 对象的原因。因此直接访问 Actor 实例是不被允许的。（actor 测试包提供了一个特殊 API，只有调用这个 API 时才能直接访问 Actor 实例。请参考 akka.testkit 包的相关介绍，http://doc.akka.io/api/akka/current/#akka.testkit.package。）

由于 ActorRef 对象非常稳定，因此这些对象之间能够组成非常稳固的依赖关系。当监管器重启 actor 时，该监管器会对 ActorRef 执行重置操作，使其指向新的 actor 实例。假如 actor 既未重启，也未恢复执行，发送到对应 ActorRef 对象的所有消息都会被转发到 ActorSystem.deadLetters 变量中，而该变量会丢弃已崩溃的 actor 所发送的消息。因此，ActorRef 对象之间的关系既稳固又可靠。

## 关于Actor的一些想法

本章出现的应用讲解了处理大规模并发输入流的一种常见模式：将具体工作分配给异步的工作节点，之后返回工作节点计算出的结果（或者像本章的示例那样，只是将这些结果打印出来）。

我们只掌握了 Akka 功能的皮毛而已。不过即便如此，你现在也已经掌握了典型的、重要的 Akka 应用，及它们的工作方式。Akka 提供了非常棒的文档，你可以访问 http://akka.io 阅读相关文档。其中附录 A 列举了一些关于 Akka 的图书。通过阅读这些图书你可以获得更深入的信息，例如：如何有效使用 Akka 的一些模式和习语。

Akka 模型实现的 actor 是轻量级的，每个 actor 大概只有 300 字节大小。因此，你可以在一个大型的 JVM 实例中轻松地创建百万个 actor 对象。对开发人员而言，如何追踪这些具有自主意识的 actor 对象是一个挑战。不过假如大多数的 actor 都是无状态的工作单元，我们就可以管理这些 actor。此外，为了满足非常高的可扩展性和可用性，你还可以使用 Akka 搭建包含上千节点的集群（http://doc.akka.io/docs/akka/current/common/cluster.html）。

对于包括 Akka 在内的 actor 模型，存在一种常见的批评，即缺乏类型安全性。我们回顾一下，`Receive` 类型是 `PartialFunction[Any,Unit]` 类型的别名，这也意味着 `Receive` 无法提供限定 actor 允许接受的消息类型的方法。因此，假如你向某个 actor 发送了一条出乎意料的消息，你必须能在运行时监测到这一问题。在逻辑正确性方面，编译器和类型系统并不会提供帮助。与之相似，actor 之间的引用都是 `ActorRef` 类型，而不是特定的某一 Actor 类型。

尽管已经有人尝试为 actor 模型添加更多的限制性输入，但是迄今还没有成功的。对于大多数用户而言，actor 模型功能强大且具备较强的灵活性，这足以弥补缺乏类型安全的缺点。

actor 模型实际上并不是纯正的函数式编程模型。`Receive` 方法返回 `Unit` 类型，这意味着在该方法中，所有事情都是通过副作用完成的！再者，只要有需要，actor 模型便会允许使用可变状态，就像我们编写的 `datastore` 那样。

不过，在使用可变状态时，需要严格遵守一条规则：将状态封装在某个 actor 中，并确保所有状态的相关操作都是线程安全的。actor 之间传递的消息应为不可变对象。不幸的是，Scala 和 Akka 自身无法保障这些可变性约束。所有这些约束都是由你自发完成的，但你可以使用一些工具来确保这些约束的正常实施。

有趣的是 actor 模型与 Alan Kay 眼中的面向对象的编程非常吻合。Alan Kay 是 Smalltalk 语言的发明人之一，也被认为是“面向对象编程”这一术语的创造者。他认为对象应该自主地对状态进行封装，这些状态只能通过消息传递的方式与其他对象通信（http://c2.com/cgi/wiki?AlanKaysDefinitionOfObjectOriented）。事实上，在 Smalltalk 语言中调用方法被称作向该对象发送了一条消息。

最后，我想说明一下，actor 模型是处理大规模、高度可用、事件驱动应用程序的更为通用的一种方法。在进一步说明这一模型之前，我们首先讨论下跨进程分配代码所面临的两个难题。我们也将一同给出这两个难题的解决方案：Pickling 库和 Spores 库。

## 17.5 Pickling 和 Spores

如何高效、可控地构造集群之间传输数据及代码的序列化和反序列机制，是分布式系统的一个挑战。这是一个古老的问题，而 Java 从发明之初便提供了一个内置的序列化机制。不过，我们还是能够构造出性能远超 Java 内嵌功能的序列化机制，而选择序列化机制时需要均衡考虑运行速度和其他的一些需求。比方说，序列化格式是否需要适用于多种语言？是否需要考虑非 JVM 语言？是否应在序列化内容中嵌入内容格式？是否需要支持版本变更？

Scala Pickling 库（https://github.com/scala/pickling）希望能够为用户提供一个序列化的方案，从而最大程度地对源代码进行简化，并提供可插拔的架构以支持不同的后台序列化格式。

之前，我们在谈论 Akka 的 `Props` 类型时曾讲过一个相关的问题：假如我们要将某一闭包（即函数字面量）分发到本进程之外的其他进程中，其他进程会捕获到什么呢？ Spores 项

目希望能够解决这个问题：Spores 会向用户提供一个 API，开发者可以通过该 API 构造一个“孢子”（即安全的闭包），其中 API 负责确保“孢子”传递的准确性。如果你希望了解 Spores 项目的更多信息或例子，请参考 Scaladoc（http://docs.scala-lang.org/sips/pending/spores.html）。

Pickling 和 Spores 项目目前均处于开发阶段，它们有望出现在 Scala 的后续版本中，也可能会作为单独的库推出。

## 17.6 反应式编程

很长时间以来，人们都认为必须使用事件驱动来处理大规模应用，这也意味着作为服务方，这些应用必须对请求做出响应；当需要获得其他服务的“帮助”时，这些应用又需要向服务提供方发送事件（或消息）。因特网就建立在这样的认知之下。由于这类系统本身是响应式的，不需要根据某些内部逻辑来执行工作，因此这类系统也被称为反应式系统。

反应式编程原理推出之后，涌现了一批模型，这些模型通过不同的方式实现了这一原理。除了 actor 模型之外，还有两个流行的模型。actor 模型认为只要能将可变状态局限在某一 actor 内，可变状态便是合理的；而这两个类型则不同，它们都比 actor 模型更为纯粹：

- *函数式反应式编程*（functional reactive programming，FRP）

  为了实现某个图像应用程序，Conal Elliott 和 Paul Hudak 使用 Haskell 语言实现了 FRP（https://wiki.haskell.org/Functional_Reactive_Programming）模型。他们实现的 FRP 是一个早期的数据流模型，在这个模型中，基于时间的状态需要通过某一系统传播到需要使用这些状态的代码中。当 FRP 模型中的某一状态发生变化时，你并不需要手动地对依赖这些变化的变量进行更新，与之相反，FRP 会使用声明的方式描述数据元素之间的依赖关系，而 FRP 运行时则会负责状态传播。因此，用户使用函数式声明语句和组合语法编写代码。最近，Evan Czaplicki 使用 Elm 语言实现了 FRP 模型，Elm（http://elm-lang.org）编写的代码能够编译成 JavaScript 代码。而名为“反对观察者模型”（Deprecating the Observer Pattern（http://lampwww.epfl.ch/~imaier/pub/DeprecatingObserversTR2010.pdf））的论文则对 Scala 语言中的一个类似的模型进行了分析。

- *反应式扩展*（reactive extensions，Rx）

  Rx（https://rx.codeplex.com）是由 Erik Meijer 和他的合作者们实现的，该模型最初只能用于 .NET 平台，随后被迁移到了许多其他语言当中，包括 Java 和 Scala（Li Haoyi（https://github.com/lihaoyi/scala.rx）实现了 Scala 的迁移工作）。Rx 模型中的可观察序列代表事件流或其他数据源。通过将可观察序列与 LINQ（LINQ 是 language-integrated query 的缩写，译为语言集成查询）库提供的查询操作符（组合器）拼接起来，Rx 组成了异步程序。

最近，已经有组织起草了 Reactive 宣言（http://www.reactivemanifesto.org/）用于提供明确的“反应式”系统定义。目前已经为反应式系统定义了四个特征。所有可伸缩、可恢复的反应式程序都应遵循这些特征。

- *消息传递或事件传递*

  反应式系统必须能对消息或事件（这些术语的定义）进行响应，这也是最基本的要求。

- *可灵活伸缩*

  为了能够满足处理要求，反应式系统是可伸缩的系统。这就意味着该系统能够通过水平扩展的方式进行扩容、调整进程数、处理核数、处理节点数。理想状态下，为了能够动态响应不停变化的处理需求，系统应该根据当前需求动态的执行水平扩展，这种调整既包括增加处理资源，也包括自动回收资源。在设计此类系统的架构时，应将网络参数（例如：性能和可靠性）视为需要考虑的头等事项。用这种方法，我们需要花费大量的精力才能对那些需要维护重要状态信息的服务进行横向扩展，而且这类系统也很难对状态信息进行“分片”和可靠地复制。

- *可恢复*

  随着系统不断变大，那些不常见的事件也会越来越频繁地出现在系统中。因此，错误也是需要考虑的头等要事。构造反应式系统时，必须不断的进行改造，以便能够在出现错误时优雅地恢复系统。

- *响应式*

  响应式系统需要能够随时对服务请求进行响应，即使系统中出现了错误的组件或是经历了非常高的流量峰值，响应式也需要通过*优雅降级*（graceful degradation）的方式继续响应用户的请求。

Actor、FRP 以及 Rx 都是基于事件的系统模型。FRP 和 Rx 模型更像是一个处理各类事件流的管道系统，而 actor 模型则像是一个帮助各个组件进行交互的网络系统。尽管略有差异，但是这些模型都能够通过多种方式进行扩展。有一点可以断言，actor 模型健壮的错误处理策略，使得它对可反应性（responsiveness）提供的支持成为最强的支持。最后再提一点，尽管这些模型提高系统响应度的方式不同，但所有模型都致力于最大程度地减少阻塞。

## 17.7 本章回顾与下一章提要

我们已经学习了如何使用 Akka 提供的 actor 为大型系统构建健壮的、可扩展的、并发应用程序。与此同时，我们还了解了 Scala 提供的进程管理功能以及 future 机制。最后，我们还讲述了反应式编程的相关概念。actor 模型便是一类反应式编程的实现。除此之外，我们还讨论了另外两个常见的模型：FRP 模型和 Rx 模型。

在一下章中，我们将会对当前最为热门的领域之一——大数据，进行讲解；同时，我们还将解释 Scala 逐渐变成大数据领域实际编程语言的原因。

# 第 18 章

# Scala与大数据

在第 17 章中，我们就讨论过编写并发程序是函数式编程被采用的推动力。然而，actor 等不错的并发模型却让开发者可以继续使用面向对象的编程技术，避免了对函数式编程的学习。所以，多核问题（multicore problem）的出现，是否并没有带来我们预期的转变呢?

现在，我认为大数据将是函数式编程更强大的推动力。尽管 actor 模型的代码看起来或多或少还是面向对象的，但使用面向对象的 Java 编写的大数据程序与使用函数式的 Scala 编写的大数据程序之间的差别是惊人的。函数式中的组合子，如：`map`、`flatMap`、`filter` 和 `fold` 等，一直是处理数据的有效工具。无论是内存集合中的小规模数据，还是分散在 PB 级规模集群里的数据，都适合采用同样的抽象。组合子的通用性对这一规模的变化几乎是无缝的。一旦你了解了 Scala 集合，你就可以迅速学会使用流行的大数据工具对应的 Scala API。的确，你最终必须理解这些工具是如何实现的以便写出更多高性能的应用，但你也可以现在就很快掌握它们。

我曾与很多有过大数据经验但从未对 Scala 产生兴趣的 Java 开发人员进行过交流。当看到采用 Scala 编写的代码可以如此简洁之后，他们感到非常高兴。正因为如此，Scala 实际上已经成为开发大数据应用的标准编程语言，至少对于像我们一样的开发者是这样的。数据科学家倾向于坚持使用自己喜欢的工具，如 R 和 Python。

## 18.1 大数据简史

大数据涵盖的工具和技术出现于过去的十年，主要用以解决三个日益凸显的挑战。第一个挑战是想方设法处理巨大的数据集。数据集的规模远大于传统方法可以管理的数据，如传统的关系数据库。第二个挑战是即使系统遇到部分失败时，也能提供持续可用性的功能。

早期的互联网巨头，如亚马逊、eBay、雅虎和谷歌，是第一批要面对这些挑战的企业。他

们每 100s 就会积累 TB 到 PB（petabyte）级别的数据，远远超任何关系型数据库和昂贵的文件存储设备的存储能力，即使现在也是如此。此外，作为互联网公司，他们需要这些数据全天 24 小时可用。即使是间隔 5~9 秒的“黄金标准”可用性也不足以满足需求。不幸的是，因为还没有学会如何应对这些挑战，许多早期的互联网公司都曾经历过尴尬和灾难性的故障。

那些成功的公司从不同的角度解决了这个问题。以亚马逊为例，亚马逊开发了一种名为 Dynamo 的数据库，它回避了关系模型，而采用简单的 key- 值存储模型，事务的支持仅限于行级别，数据则分片存储在一个集群中（在著名的 Dynamo 研究论文（http://www.cs.ucsb.edu/~agrawal/fall2009/dynamo.pdf）中有描述）。作为回报，它们得以通过水平扩展存储大量的数据，具有高的读写吞吐量，并具有更高的可用性。由于复制策略，节点和机架故障不会导致数据丢失。当今许多流行的 NoSQL 数据库都受到了 Dynamo 的启发。

谷歌开发了一个集群、虚拟的文件系统，称为谷歌文件系统（http://research.google.com/archive/gfs.html）（Google File System，GFS），该系统拥有类似 Dynamo 的可扩展性和可用性。在 GFS 之上，他们建立了一个通用的计算引擎，将分析工作分发到集群中。由于任务运行在多个节点上，从而充分利用了并行，实现比单线程程序更快的数据处理。这个称为 MapReduce（https://www.usenix.org/legacy/event/osdi04/tech/full_papers/dean/dean.pdf）的计算引擎，它引发了从类似 SQL 的查询到机器学习算法等一大批应用的出现。

GFS 和 MapReduce 启发了一个克隆实现，它们合起来被称为 Hadoop（http://hadoop.apache.org）。因为其他许多公司也开始使用 Hadoop 来存储和分析自己不断增长的数据集，Hadoop 在 21 世纪后期得到迅速应用。它的文件系统被称为 HDFS：Hadoop Distributed File System（Hadoop 分布式文件系统）。

现在，具有大数据集的组织往往同时部署了 Hadoop 和 NoSQL 数据库，以支持它们的众多应用，从成本较低的数据仓库、其他“离线”分析，到超大型的事务处理等。

“大数据”这个词也有些许不当，因为许多数据集并没有那么大。但人们发现通过使用灵活、低成本的大数据工具实现存档、集成与分析数据的方式非常有用，而且这些数据格式多样、来源广泛。

本章的其余部分将主要讨论大数据的计算引擎，探索 MapReduce 工具的演变过程以及 MapReduce 如何被更好的工具代替。在这个过程中，Scala 就处于这种演变的前沿和中心的位置。

对于大多数 NoSQL 数据库，Scala 都有相应的 API；而且在大多数情况下，这些 API 的设计都符合惯例，与你可能使用过的 Java API 很类似。这样，我们就可以集中精力于更难的问题上，而不是把时间花在广泛普及的概念上。这些较难的问题包括简化函数式编程和增强以数据为中心的应用程序。

## 18.2 用Scala改善MapReduce

MapReduce 的 Java API 是非常底层的但很难使用，它需要特别的专业知识来实现一些算法，才能获得良好的性能。MapReduce 模型包含 map 步骤，在该步骤中我们通过读

入文件，将数据转为 key 值对的形式，来满足算法的要求。key 值对在 shuffle 阶段被混洗，安排相同的 key 在一起，然后执行最后的 reduce 处理步骤。许多算法都需要好多个 MapReduce 的 job 串在一起。不幸的是，MapReduce 在每个 job 结束后，会将数据刷到磁盘中，即使该序列中的下一个作业需要将数据读回内存。反复执行磁盘 I/O 是 MapReduce 处理大型数据集时效率不高的主要原因。

在 MapReduce 中，map 其实表示平坦映射（flat map），这是因为对于每一个输入（比如，文本文件中的一个）会产生零到多个输出的 key 值对。Reduce 的含义与通常认为的相同。但是，试想一下，如果 Scala 的容器只有 flatMap 和 reduce 这两个组合子，会如何呢？许多你想完成的转换会很难实现。另外，你需要了解如何在大型数据集上有效地实现转换。其结果如下：在原则上，你可以在 MapReduce 实现几乎所有的算法，但在实践中，这需要特殊的专业知识和具有挑战性的编程工作。

Cascading（http://cascading.org）是 Java 中最有名的 API，它支持对 Hadoop MapReduce 中的典型数据问题进行抽象，也隐藏了许多底层的 MapReduce 细节（请注意，在我写这本书时，用于消除 MapReduce 的替代后端工具正在开发中）。Twitter 发明了 Scalding（https://github.com/twitter/scalding）。Scalding 是基于 Cascading 的 Scala API，已经变得非常流行。

下面我们来看一个典型的算法 Word Count，它是 Hadoop 的“Hello World”。因为它的概念很容易理解，你可以专注于学习 API。在 Word Count 中，文档语料由并行的 map 任务（通常每个任务读取一个文件）读入。文本被拆分为单词，每个任务输出（word，count）对组成的序列，其中 count 是 word 在该文档中出现的次数。在最简单的一种实现中，map 每当遇见 word，就输出（word，1）。不过优化性能的话，map 对每个单词只输出一个（word，count）对，这样可以减少输送给 reduce 的 key 值对个数。在这个算法中，word 是 key。

混洗进程将所有相同的 word 元组组合到一起并分配给同一个 reduce 任务，在 reduce 中计算出单词的最终出现次数，并把所有的结果写回磁盘。在逻辑上，写 10 次（*Foo*，*1*）元组与写 1 次（*Foo*，*10*）元组是一样的，因为加法满足结合律，在任何阶段进行相加都一样。

下面我们来比较以下 3 种 API 实现 Word Count：Java MapReduce、Cascading 和 Scalding。

> 由于这些例子需要其他一些依赖才能编译执行，部分依赖的工具包还不支持 Scala 2.11，工具包里的文件都带“X”后缀，因此 `sbt` 不会编译它们。脚注中包含每个例子如何编译运行的信息。

为节省篇幅，我只会给出 Hadoop MapReduce 版本的部分代码。完整的代码可以从本书的随书代码实例中下载，地址在注释中给出：[1]

```
// src/main/java/progscala2/bigdata/HadoopWordCount.javaX
...
class WordCountMapper extends MapReduceBase
```

注 1：Hadoop 教程中有另一种实现，同时还有编译、运行 Hadoop 程序的命令说明。另外也可参见 Tom White 的《Hadoop 权威指南（第 3 版）》来获取相关的知识。

```
    implements Mapper<IntWritable, Text, Text, IntWritable> {

  static final IntWritable one  = new IntWritable(1);
  static final Text word = new Text;

  @Override public void map(IntWritable key, Text valueDocContents,
      OutputCollector<Text, IntWritable> output, Reporter reporter) {
    String[] tokens = valueDocContents.toString.split("\\s+");      // ❶
    for (String wordString: tokens) {
      if (wordString.length > 0) {
        word.set(wordString.toLowerCase);
        output.collect(word, one);
      }
    }
  }
}

class WordCountReduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

  public void reduce(Text keyWord, java.util.Iterator<IntWritable> counts,
      OutputCollector<Text, IntWritable> output, Reporter reporter) {
    int totalCount = 0;
    while (counts.hasNext) {                                         // ❷
      totalCount += counts.next.get;
    }
    output.collect(keyWord, new IntWritable(totalCount));
  }
}
```

❶ map 步骤的实际工作。文本被切分为单词，每个单词都作为 key 写到输出中，同时写入的还有数字 1，作为 key 值。

❷ reduce 步骤的实际工作。对于每个 key（即单词），计算其值的总和。单词及其出现次数的总和被写入输出，保存到文件中。

这段代码让我想起了原来的 EJB 1.X API。侵入性非常高，也不够灵活。样板代码比比皆是。你必须将所有字段包装在序列化格式 `Writable` 中，因为 Hadoop 不会为你做这些。Java 的惯用方法实现起来非常繁琐。提供 `main` 函数的代码没有给出，其中添加了 12 行左右的代码，用来设置应用程序。我不会解释所有的 API 细节，但希望你已经理解到，这里所用的 API 需要注意很多与算法本身毫无关系的细节。

除去 import 语句和注释，这个版本约有 60 行代码。这不算多。MapReduce 的 job 通常比通用的 IT 应用要小，不过这是因为这个算法非常简单。要实现更复杂的算法，复杂程度会显著提高。例如：你可能会写一个常见的 SQL 查询语句，数据保存在一个名为 `wordcount` 的表中：

```
SELECT word, count FROM wordcount ORDER BY word ASC, count DESC;
```

很简单。在互联网上搜索 secondary sort mapreduce，然后你会找到 MapReduce 的实现，其复杂度让人感到惊讶。

Cascading 提供了一个直观的管道（pipe）模型，管道被连接到 flow，其中的数据源和水槽

是 tap。下面的完整代码（省略了 import 语句）是等价的 Cascading 版本[2]的实现：

```
// src/main/java/progscala2/bigdata/CascadingWordCount.javaX
package impatient;

import ...;

public class CascadingWordCount {
  public static void main( String[] args ) {
    String input  = args[0];
    String output = args[1];

    Properties properties = new Properties();                    // ❶
    AppProps.setApplicationJarClass( properties, Main.class );
    HadoopFlowConnector flowConnector = new HadoopFlowConnector( properties );

    Tap docTap = new Hfs( new TextDelimited( true, "\t" ), input );  // ❷
    Tap wcTap = new Hfs( new TextDelimited( true, "\t" ), output );

    Fields token = new Fields( "token" );                        // ❸
    Fields text = new Fields( "text" );
    RegexSplitGenerator splitter =
        new RegexSplitGenerator( token, "[ \\[\\]\\(\\),.]" );
    Pipe docPipe =                                               // ❹
      new Each( "token", text, splitter, Fields.RESULTS );

    Pipe wcPipe = new Pipe( "wc", docPipe );                     // ❺
    wcPipe = new GroupBy( wcPipe, token );
    wcPipe = new Every( wcPipe, Fields.ALL, new Count(), Fields.ALL );

    // 将所有的tap、pipe等连接到flow。
    FlowDef flowDef = FlowDef.flowDef()                          // ❻
     .setName( "wc" )
     .addSource( docPipe, docTap )
     .addTailSink( wcPipe, wcTap );

    // 运行flow。
    Flow wcFlow = flowConnector.connect( flowDef );              // ❼
    wcFlow.complete();
  }
}
```

❶ 少量用于设置的代码，包括为 Hadoop 的运行做的配置。

❷ 用 HDFS 的 tap 读写数据。

❸ 定义元组中的两个字段，以表示一条记录。用正则表达式将文本切分为单词组成的流。

❹ 创建一个管道，用来迭代输入的文本，输出单词。

❺ 创建新管道，对单词执行分组操作，单词作为分组的 key。然后用另一个管道计算每一组的单词数。

注 2：改编自 *Cascading for the Impatient*，Part 2（http://docs.cascading.org/impatient），© 2007-2013 Concurrent，Inc. 版权所有。

❻ 创建 flow，并输入输出连接到管道中。

❼ 运行该程序。

除去 import 语句，Cascading 版本的代码大约有 30 行。即使对该 API 不太了解，也能理解该算法。将文本切分为单词后，我们对单词进行分组，然后对每一组计算个数。这就是算法所做的工作。如果我们的数据表里有“原始的单词”的话，对应的 SQL 查询语句会是这样：

```
SELECT word, COUNT(*) as count FROM raw_words GROUP BY word;
```

Cascading 提供了一组优雅的 API，该 API 已变得非常流行。但 Cascading 在开发时，受到了 Java 相对冗长的语法和 Java 8 以前缺少匿名函数等缺点的限制。例如：`Each`、`GroupBy` 及 `Every` 等对象应该用高阶函数实现。Scalding 就是这么做的。

以下就是 Scalding 的版本[3]：

```
// src/main/scala/progscala2/bigdata/WordCountScalding.scalaX

import com.twitter.scalding._                                    // ❶

class WordCount(args : Args) extends Job(args) {

  TextLine(args("input"))                                        // ❷
    .read
    .flatMap('line -> 'word) {                                   // ❸
      line: String => line.trim.toLowerCase.split("""\s+""")
    }
    .groupBy('word){ group => group.size('count) }               // ❹
    .write(Tsv(args("output")))                                  // ❺
}
```

❶ 只有一个重要的 import 语句。

❷ 读进文件，其中的每一行都是一条“记录”。`TextLine` 抽象了本地文件系统、HDFS、S3 等。Scalding 运行的方式决定文件系统的路径的解释方式，在这一点上，Cascading 却要求你在代码中来决定。每一行字段名均为 `'line`，Scalding 用 Scala 符号来指定字段名。表达式 `args("input")` 表示从命令行参数 `--input` *`path`* 中获取路径。

❸ 得到每个 `'line`，用 `flatMap` 将其切分为单词。语法 `('line -> 'word)`，表示我们提取输入的一个字段（当前只有一个字段），输出的字段被称为 `'word`。

❹ 以单词为 key 进行分组，然后计算组的大小。输出的记录形式为 `('word, 'count)`。

❺ 将输出以 Tab 分隔的形式写入到命令行参数 `--output` *`path`* 指定的路径中去。

Scalding 的版本代码只有十几行，还包括了 import 语句！现在，几乎所有的架构细节退居幕后。这里纯粹是算法。你能知道 `flatMap` 和 `groupBy` 在做什么，即使 Scalding 为大多数组合子添加了一个额外的参数列表，用于字段选择。

注 3：改编自 GitHub 上的 scalding-workshop 示例。

我们从冗长、繁琐的程序演变为一个简单的脚本。当你可以写出如此简洁的程序的时候，整个软件开发过程都改变了。

## 18.3 超越MapReduce

在“实际的时间”处理事件的需求正在成为趋势。MapReduce 只适用于完成批处理模式。HDFS 在最近才增加了对文件增量更新的支持。大多数的 Hadoop 工具还不支持此功能。

这种趋势使得新的工具被创造出来，如：storm（http://storm.apache.org/），该工具是一个集群事件处理系统。

其他日益受到关注的是 MapReduce 的性能限制，如前面提到的过度磁盘 I/O，以及 API 和底层模型上遇到的困难。

大多数第一代技术都有一定的局限性，最终导致它们被其他技术替代。Hadoop 的主要厂商最近接受了 MapReduce 的替代者，即 Spark（http://spark.apache.org），它同时支持批处理模式和流模式。Spark 用 Scala 编写而成，相比 MapReduce，它提供了出色的性能，部分原因是它将数据缓存在了步骤间的内存中。也许最重要的是，Spark 提供了类似 Scalding 提供的直观的 API，但 Spark 提供的 API 具有令人难以置信的简洁性和丰富的表现力。

Scalding 和 Cascading 使用管道，而 Spark 使用弹性分布数据集（resilient、distributed dataset，RDD），这是一种分布在集群的内存中的数据结构。它的弹性是指，如果一个节点出现故障，Spark 知道如何从数据源中重建缺失的这一块。

以下是 Word Count 用 Spark 的实现[4]：

```
// src/main/scala/progscala2/bigdata/WordCountSpark.scalaX
package bigdata

import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object SparkWordCount {
  def main(args: Array[String]) = {
    val sc = new SparkContext("local", "Word Count")              // ❶
    val input = sc.textFile(args(0)).map(_.toLowerCase)           // ❷
    input
      .flatMap(line => line.split("""\W+"""))                     // ❸
      .map(word => (word, 1))                                     // ❹
      .reduceByKey((count1, count2) => count1 + count2)           // ❺
      .saveAsTextFile(args(1))                                    // ❻
    sc.stop()                                                     // ❼
  }
}
```

❶ 首先是 `SparkContext`。第一个参数用来指定“主”节点，在这个例子中，我们在本地运行。第二个参数用来为任务指定一个任意的名字。

注 4：改写自 GitHub 上的 spark-workshop 示例。

❷ 从第一个命令行参数指定的路径中加载一个或多个文本文件（在 Hadoop 中，我们给出目录的路径，然后目录下所有的文件都会被读取），然后将字符串转为小写，返回一个 RDD。

❸ 以非字母数字的字符为分隔符，将每行文本映射为一到多个单词。

❹ 将每个单词映射为一个元组（`word, 1`）。重新调用用于 Word Count 的 Hadoop map 的输出结果。

❺ 先使用 `reduceByKey`，其功能与 SQL 语句 `GROUP BY` 类似，然后进行归约（reduction）。在这里，归约就是将元组里的多个 1 相加。在 Spark 中，元组的第一个元素默认为操作中的 key，元组剩下的元素为 key 的值。

❻ 将结果写入第二个输入参数指定的路径中。Spark 遵循 Hadoop 的惯例，将路径作为目录，它会将每个最终任务产生的结果写入各自的文件中（命名约定是 `part-n`，`n` 是五位数字，从 `00000` 开始）。

❼ 关闭上下文，停止运行。

类似于 Scalding 的例子，这个程序也大约只有十几行代码。

无论你使用比较成熟、但仍在成长的工具（如 Scalding），或者使用刚刚崭露头角的工具（如 Spark），很明显，Scala 的 API 有一个超越基于 Java 的 API 的独特优势。我们已经熟知的函数式组合子，是用于数据分析的理想工具，对这些工具的用户和实现者来说都是如此。

## 18.4　数学范畴

我们在第 16 章讨论了范畴。有一个范畴 Monoid，当初没有讨论到但在大数据中逐渐流行起来。如果没有仔细阅读第 16 章，你只要认为范畴就是面向数学的设计模式即可。

Monoid 是加法的抽象。它具有以下特性。

(1) 它是一个满足结合律的二元操作符。
(2) 有单位元素。

数字的加法满足以下性质。结合律：(1.1 + 2.2) + 3.3 == 1.1 + (2.2 + 3.3)；0 是加法的单位元素。乘法也是如此，1 是乘法的单位元素。数字的加法和乘法同时也满足交换律：1.1 + 2.2 == 2.2 + 1.1，不过这对 Monoid 来说不是必需的。

这有什么大不了的？事实证明，大量的数据结构都满足这些特性，所以如果你让代码符合 Monoid 的性质，代码将是高度可复用的（参见维基百科上的列表，https://en.wikipedia.org/wiki/Monoid）。

例子包括有字符串连接、矩阵加法和乘法、计算最大值和最小值、以及一些近似算法，如用于计算基数的 HyperLogLog 算法，求相似性的 Min-hash 算法以及用于设置成员关系的布隆过滤器（参见 Avi Bryant 的精彩演讲 *Add ALL The Things!*，http://www.infoq.com/presentations/abstract-algebra-analytics）。

部分数据结构也满足交换律，所有数据结构都可以对大数据集实行并行的执行。这些近似

算法用较差的精度换取更高的空间效率。

你会在很多数学相关的包中看到 Monoid 的实现，这些包括下一节列出的数据。

## 18.5 Scala数据工具列表

除了 Hadoop 的平台和 Scala 的数据库 API，还出现了很多工具用于处理数据相关的问题，如一般的数学问题和机器学习问题。表 18-1 列出了你有可能会查阅的一些活跃项目，为完整起见，这里也包括我们先前讨论过的。

表18-1：数据和数学库

| 选　项 | URL | 描　述 |
|---|---|---|
| Algebird | http://bit.ly/10Fk2F7 | 推特的抽象代数 API，可与几乎所有大数据 API 配合 |
| Factorie | http://factorie.cs.umass.edu/ | 一个用于部署概率模型的工具箱，用简洁的语言创建关系的因素图表，估计参数，并进行推断 |
| Figaro | http://bit.ly/1nWnQf4 | 一个用于概率相关编程的工具箱 |
| H2O | http://bit.ly/1G2rfz5 | 一个用于数据分析的高性能、基于内存的分布式计算引擎，用 Java 和 Scala、R 语言的 PAI 写成 |
| Relate | http://bit.ly/13p17zp | 专注性能的数据库轻量级访问层 |
| ScalaNLP | http://www.scalanlp.org/ | 一套机器学习和数值计算库。这是一个总体项目数库，包括 Breeze，它用于机器学习和数值计算，以及 Epic，它用于进行统计分析和结构化预测 |
| ScalaStorm | http://bit.ly/10aaroq | Scala 的 Storm API |
| Scalding | https://github.com/twitter/scalding | Twitter 用于 Cascading 的 API，用于将 Scala 推广为一门 Hadoop 编程语言 |
| Scoobi | https://github.com/nicta/scoobi | 对 MapReduce 的顶层抽象层，API 类似 Scalding 和 Spark 的 API |
| Slick | http://slick.typesafe.com/ | Typesafe 开发的数据库访问层 |
| Spark | http://spark.apache.org/ | 在 Hadoop 环境中进行分布式计算的框架，正在成长为业界标准。也能用在 Mesos 集群和单个设备（“本地”模式）上 |
| Spire | https://github.com/non/spire | 以通用、快速、精确为目的的数值库 |
| Summingbird | https://github.com/twitter/summingbird | Twitter 的 API，它将 Scalding（批处理模式）和 Storm（事件流）的计算抽象出来 |

尽管 Hadoop 环境得到了很多的关注，但通用的工具，如：Spark、Scalding/Cascading 和 H2O 在不必要使用大的 Hadoop 集群时，依然支持较小规模的部署。

## 18.6 本章回顾与下一章提要

在我们这个行业，几乎没有什么分支比大数据更能促进 Scala 的发展。Scalding 和 Spark 对 Java 的 MapReduce API 的改善十分惊人，甚至是颠覆性的。两者使得 Scala 成为以数据为

中心的应用程序开发的不二之选。

通常情况下，我们认为 Scala 是一个像 Java 那样的静态类型语言。但是，Scala 标准库中包含了一个特殊的 trait，用来创建具有更多动态行为的类型，就像你在 Ruby 和 Python 之类的语言中看到的动态行为一样。这些动态行为将在下一章中进行介绍。Scala 的这一特殊的特性是用于构建领域特定语言（DSL）的工具，我们也将在下一章讨论到。

# 第 19 章

# Scala动态调用

大多数时候，Scala 静态类型能够给代码带来好处。静态类型引入了一些安全约束，这有助于确保运行时的正确性，而阅读代码时，静态类型也更易于理解。当处理大型系统时，这些特点尤为有用。

尽管使用动态系统能够让你调用一些编译时尚不存在的方法，但是有时候，你也许都忘记了动态类型能给我们带来哪些好处。著名的 Ruby On Rails web 框架（http://rubyonrails.org）的 `ActiveRecord` API 很好地应用了这项技术。接下来，我们将学习如何在 Scala 中实现相同的技术。

## 19.1 一个较为激进的示例：Ruby on Rails框架中的ActiveRecord库

`ActiveRecord` 库是 Rail 框架中最初集成的对象关系映射（ORM）库。尽管对于 `ActiveRecord` 的大多数细节我们并不关心[1]，不过该库所提供的一门用于组合查询的 DSL 语言却是例外。使用这门 DSL 语言可以对领域对象执行链式方法。

不过，`ActiveRecord` 库中实际上并未定义这些“方法”。和 Ruby 中所有未定义的方法调用一样，这些方法调用会被“分发”给 `method_missing` 方法。通常，该方法会直接抛出异常，不过我们也可以在类中对该方法进行覆写以执行其他操作。`ActiveRecord` 便是通过这种方式将这些“不存在的方法”解释成用于构造 SQL 查询的指令的。

注 1：如果您希望了解更多关于 `ActiveRecord` 库的细节，请查看 Active Record Basics（http://guides.rubyonrails.org/active_record_basics.html）。

假设我们定义了一个简单的数据库表，该表中记录了美国各州的基本信息（我们使用某种 SQL 方言创建该表）：

```
CREATE TABLE states (
  name       TEXT,     -- 州名
  capital    TEXT,     -- 州府所在城市名
  statehood  INTEGER   -- 该州加入美联邦的年份
);
```

使用 ActiveRecord 库时，你可以通过下列方式构造查询，下文中出现的 Ruby 领域对象 State 等同于 states 表：

```
# 找到所有名为"Alaska"的州
State.find_by_name("Alaska")
# 找到所有名为"Alaska"且在1959年加入美联邦的州
State.find_by_name_and_statehood("Alaska", 1959)
...
```

如果数据表中包含了大量的列，想要定义所有的 find_by_* 方法组合是不可能的。不过，由于这些方法的命名规范确保了这些方法很容易自动生成，因此我们无需定义这些方法。ActiveRecord 库自动完成了方法名称解析、生成对应的 SQL 查询以及构造查询结果对应的内存对象这些枯燥的工作。

值得注意的是，ActiveRecord 实现了一种嵌入式 DSL，或者称为内部 DSL。这门 DSL 语言是宿主语言 Ruby 的一类惯用方言，它并不是需要自定义语法和解析器的另外一门语言。

## 19.2　使用动态特征实现Scala中的动态调用

如果我们能够在 Scala 中实现类似的 DSL 语言，这也许会带来一些好处。不过通常情况下，Scala 要求这类方法事先需要显式地定义。幸运的是，为了能够支持我们刚才描述的动态特征，Scala 2.9 已经添加了 scala.Dynamic（http://www.scala-lang.org/api/current/index.html#scala.Dynamic）特征。

Dynamic 特征只起到了标示作用，该 trait 中并不包含任何方法定义。假如编译器看到了这一 trait，在处理该 trait 时会遵循某一特殊协议。Dynamic 特征的 Scaladoc 页面（http://www.scala-lang.org/api/current/index.html#scala.Dynamic）中对该协议进行了概述，并使用了下面的示例进行了讲解。在该示例中，foo 对象是 Foo 类的实例，而 Foo 类则继承了 Dynamic 特征：

```
foo.method("blah")      ~~> foo.applyDynamic("method")("blah")
foo.method(x = "blah")  ~~> foo.applyDynamicNamed("method")(("x", "blah"))
foo.method(x = 1, 2)    ~~> foo.applyDynamicNamed("method")(("x", 1), ("", 2))
foo.field               ~~> foo.selectDynamic("field")
foo.varia = 10          ~~> foo.updateDynamic("varia")(10)
foo.arr(10) = 13        ~~> foo.selectDynamic("arr").update(10, 13)
foo.arr(10)             ~~> foo.applyDynamic("arr")(10)
```

Foo 类型必须实现可能会被调用的所有动态方法。applyDynamic 方法作用于未使用命名参数的方法调用。假如用户给某些参数命名了，那么调用该方法时将会调用

applyDynamicNamed 方法。请注意，第一个参数列表中只包含了单一参数，该参数代表了将要调用的方法名称。而第二个参数列表中则包含了需要传递给对应方法的实际参数。

假如希望声明特定的类型化参数集，你可以通过声明第二参数列表的方式以支持可变数量的参数。这完全取决于你希望用户以何种方式调用这些方法。

我们可以使用 selectDynamic 和 updatDynamic 方法对非数组类型的字段进行读写操作。从第二个例子开始到最后一个例子都展示了编写数组元素操作时所需要使用的特殊格式。读取数组元素时调用的方法中包含多个参数。因此，读取数组元素时我们应使用 applyDynamic 方法。

我们将使用 Dynamic 特征为 Scala 构造一门简单的查询 DSL 语言。实际上，我们的示例与 .NET 语言中被称为 LINQ（http://msdn.microsoft.com/en-us/library/bb397926.aspx）（集成语言查询，是 language-integrated query 的简称）的查询 DSL 非常接近。LINQ 将类 SQL 查询嵌入到 .NET 程序中，在操作 LINQ 时，你可以使用集合、数据库表等对象。Scala 的*函数式关系映射*（functional-relational mapping，FRM）库 Slick（http://slick.typesafe.com）便借鉴了 LINQ 的思想。

由于只实现一小部分的可能操作，因此我们将这些实现称为 CLINQ，意思是*廉价版集成语言查询*（cheap language-integrated query）。我们将定义一个名为 CLINQ 的 case 类，当然，这个类名听起来确实有点傻。

假设我们希望对内存中的数据结构进行查询，尤其是希望能够使用借鉴了 SQL 思想的 DSL 对一系列 Map 对象（key 值对）执行操作。除了提供示例代码之外，我们还提供了实现代码。我们将首先执行下列脚本，一方面对我们所需要使用的 DSL 语法进行演示，另一方面也能证明我们的实现工作的正常运行。

```
// src/main/scala/progscala2/dynamic/clinq-example.sc

scala> import progscala2.dynamic.CLINQ
import progscala2.dynamic.CLINQ

scala> def makeMap(name: String, capital: String, statehood: Int) =
     |   Map("name" -> name, "capital" -> capital, "statehood" -> statehood)

// 记录了美国州信息的五条“记录”。
scala> val states = CLINQ(
     |   List(
     |     makeMap("Alaska",     "Juneau",      1959),
     |     makeMap("California", "Sacramento",  1850),
     |     makeMap("Illinois",   "Springfield", 1818),
     |     makeMap("Virginia",   "Richmond",    1788),
     |     makeMap("Washington", "Olympia",     1889)))
states: dynamic.CLINQ[Any] =
Map(name -> Alaska, capital -> Juneau, statehood -> 1959)
Map(name -> California, capital -> Sacramento, statehood -> 1850)
Map(name -> Illinois, capital -> Springfield, statehood -> 1818)
Map(name -> Virginia, capital -> Richmond, statehood -> 1788)
Map(name -> Washington, capital -> Olympia, statehood -> 1889)
```

在本示例中，我们导入了 dynamic.CLINQ 这一 case 类。稍后我们将学习如何实现这一 case

类。我们之后创建了一个包含了一组 map 的实例，每个 map 对象都代表了一个“记录”。

与 ActiveRecord 示例不同，为了获取希望的字段数据（如 SQL SELECT 语句），我们将使用像 n_and_m 这样的方法执行投影操作，而 all 方法则对应了 SELECT * （我们将省略某些输出结果）语句：

```
scala> states.name
res0: dynamic.CLINQ[Any] =
Map(name -> Alaska)
Map(name -> California)
Map(name -> Illinois)
Map(name -> Virginia)
Map(name -> Washington)

scala> states.capital
res1: dynamic.CLINQ[Any] =
Map(capital -> Juneau)
Map(capital -> Sacramento)
...

scala> states.statehood
res2: dynamic.CLINQ[Any] =
Map(statehood -> 1959)
Map(statehood -> 1850)
...

scala> states.name_and_capital
res3: dynamic.CLINQ[Any] =
Map(name -> Alaska, capital -> Juneau)
Map(name -> California, capital -> Sacramento)
...

scala> states.name_and_statehood
res4: dynamic.CLINQ[Any] =
Map(name -> Alaska, statehood -> 1959)
Map(name -> California, statehood -> 1850)
...

scala> states.capital_and_statehood
res5: dynamic.CLINQ[Any] =
Map(capital -> Juneau, statehood -> 1959)
Map(capital -> Sacramento, statehood -> 1850)
...

scala> states.all
res6: dynamic.CLINQ[Any] =
Map(name -> Alaska, capital -> Juneau, statehood -> 1959)
Map(name -> California, capital -> Sacramento, statehood -> 1850)
...
```

那么，我们又该如何实现 WHERE 子句的功能呢？

```
scala> states.all.where("name").NE("Alaska")
res7: dynamic.CLINQ[Any] =
Map(name -> California, capital -> Sacramento, statehood -> 1850)
```

```
Map(name -> Illinois, capital -> Springfield, statehood -> 1818)
Map(name -> Virginia, capital -> Richmond, statehood -> 1788)
Map(name -> Washington, capital -> Olympia, statehood -> 1889)

scala> states.all.where("statehood").EQ(1889)
res8: dynamic.CLINQ[Any] =
Map(name -> Washington, capital -> Olympia, statehood -> 1889)

scala> states.name_and_statehood.where("statehood").NE(1850)
res9: dynamic.CLINQ[Any] =
Map(name -> Alaska, statehood -> 1959)
Map(name -> Illinois, statehood -> 1818)
Map(name -> Virginia, statehood -> 1788)
Map(name -> Washington, statehood -> 1889)
```

CLINQ 完全不知道 map 对象中存储了哪些 key 值，不过利用 Dynamic 特征，我们能够根据这些 key 值构造出对应的方法。下面列出了对应的 CLINQ 代码。

```
// src/main/scala/progscala2/dynamic/CLINQ.scala
package progscala2.dynamic
import scala.language.dynamics                                   // ❶

case class CLINQ[T](records: Seq[Map[String,T]]) extends Dynamic {

  def selectDynamic(name: String): CLINQ[T] =                    // ❷
    if (name == "all" || records.length == 0) this               // ❸
    else {
      val fields = name.split("_and_")                           // ❹
      val seed = Seq.empty[Map[String,T]]
      val newRecords = (records foldLeft seed) {
        (results, record) =>
          val projection = record filter {                       // ❺
            case (key, value) => fields contains key
          }
          // 终止没有投影操作的记录。
          if (projection.size > 0) results :+ projection
          else results
      }
      CLINQ(newRecords)                                          // ❻
    }

  def applyDynamic(name: String)(field: String): Where = name match {
    case "where" => new Where(field)                             // ❼
    case _ => throw CLINQ.BadOperation(field, """Expected "where".""")
  }

  protected class Where(field: String) extends Dynamic {         // ❽
    def filter(value: T)(op: (T,T) => Boolean): CLINQ[T] = {     // ❾
      val newRecords = records filter {
        _ exists {
          case (k, v) => field == k && op(value, v)
        }
      }
      CLINQ(newRecords)
    }
```

```
    def applyDynamic(op: String)(value: T): CLINQ[T] = op match {
      case "EQ" => filter(value)(_ == _)                          // ❿
      case "NE" => filter(value)(_ != _)                          // ⓫
      case _ => throw CLINQ.BadOperation(field, """Expected "EQ" or "NE".""")
    }
  }

  override def toString: String = records mkString "\n"          // ⓬
}

object CLINQ {                                                   // ⓭
  case class BadOperation(name: String, msg: String) extends RuntimeException(
    s"Unrecognized operation $name. $msg")
}
```

❶ Dynamic 特征是可选的语言特性，为了使用它，我们需要导入该 trait。

❷ 我们将使用 selectDynamics 方法执行字段投影操作（projection）。

❸ 假如传入了 all 这一“关键字”，或者内存中没有任何记录，将返回所有的字段信息。

❹ 由于 CLINQ 使用 _and_ 连接词将两个或多个字段连接起来，因此我们需要将方法名分解成一组字段名称。

❺ 对 map 对象进行过滤，只返回名字中包含的字段。

❻ 构造并返回新的 CLINQ 对象。

❼ 执行投影操作后，我们将调用 applyDynamic 方法。执行 applyDynamic 方法后将得到一个新的 Where 类实例，Where 类也继承自 Dynamic 特征。请注意，由于 Where 实例包含了相同记录集中的某些记录，因此我们无需构造新的记录集对象！假如你传入了其他的类 SQL 的关键字，applyDynamic 方法便会抛出错误。

❽ Where 类会根据 field 字段值，对记录集进行过滤。

❾ filter 是一个辅助方法，它会对记录集里的记录进行过滤，filter 方法会选择那些 key 值对中键名与指定的 field 参数值相同的 Map 对象，根据该 Map 对象的 key 值和对应的 value 值 v 执行操作 op(value, v)，只有操作返回 true 的记录才会被返回。

❿ 假如向 applyDynamic 方法传入 EQ 操作符，该方法将调用 filter 方法对记录集进行过滤，只有那些指定字段的数值与用户指定值相等的记录才会被返回。

⓫ applyDynamic 方法对不等于操作提供了支持。请注意，由于并不是所有可能的值类型都支持像大于、小于这样的操作，因此处理这些类型时需要格外仔细。

⓬ 根据记录信息，创建易读的字符串。

⓭ 在伴生对象中定义了 BadOperation 异常。

当然，从很多方面来看，CLINQ 都只是一个穷人版的 LINQ 实现。它没有实现 SQL 中的其他有用的操作，如 GROUP BY 操作。同时它提供的 WHERE 子句操作也没有实现像大于、小于这样的功能。由于并不是所有的值类型都支持这些操作，因此需要使用一些技巧才能使 CLINQ 支持这些操作，不过这些操作并非真的无法实现。

## 19.3 关于DSL的一些思考

`Dynamic` 特征是 Scala 用于实现嵌入式 DSL 和内部 DSL 的众多工具中的一件工具。我们将在下一章中更深入地学习这些工具。现在，需要说明一些注意事项。

首先，利用 `Dynamic` 特征实现的代码并不容易被人理解，这意味代码维护、调试以及扩展都会比较困难。使用像 `Dynamic` 这样很酷的工具是很有吸引力的，但是使用之后你需要花费大量的时间来打理代码，这会让人非常懊悔。因此，如果你希望使用 `Dynamic` 以及其他的一些 DSL 功能，请明智地判断是否应该使用这些功能。

其次，有一个与 DSL 相关的难题折磨着所有的 DSL 开发者。这个挑战便是如何为用户提供有意义且有用的错误信息？我们以上一节的示例为例，很容易写出像编译器无法解析这样的错误信息，而这样的错误信息并不能为用户提供太大的帮助。（提示：编写 DSL 时可以尝试使用中缀表达式，这样能够省略掉一些点号和括号。）

再次，好的 DSL 需要防止用户编写一些不合逻辑的东西。本章列举的简单示例并不会遇到这种难题，不过对于一些更为高级的 DSL 而言，这确实是一个挑战。

## 19.4 本章回顾与下一章提要

我们已经学习了如何通过 Scala 提供的“钩子”来编写出包含动态方法和动态值的代码。那些使用像 Ruby 这样的动态语言的用户对这些动态类型都已经非常熟悉了。我们使用这些动态类型实现了一门查询 DSL 语言，这门 DSL 语言能够根据输入的数据值，像变魔术一样为这些值生成对应的方法！

不过，在编写 DSL 时，我们所使用的像 `Dynamic` 这样的功能会为我们带来一些挑战。幸运的是，Scala 为我们提供了一些可用于编写 DSL 语言的工具，我们也将在下一章对这部分内容进行深入学习。

# 第20章

# Scala的领域特定语言

领域特定语言（domain-specific language，DSL）是一门模仿特定领域专业人员所熟知的术语、习惯用法和表达方式的编程语言。DSL 的代码读起来像是相应领域术语内结构化的散文。理想情况下，一个领域的专业人员可以在几乎没有编程经验或者不能编写 DSL 代码的情况下，阅读、理解和验证 DSL 代码。

我们只会简单介绍 DSL 这个大课题及 Scala 对 DSL 的支持。想要了解更深入的知识，请参阅附录 A 中的 DSL 部分。

精心设计的 DSL 有以下几个优点。

- *封装性好*

  DSL 隐藏了实现细节，只暴露那些与领域相关的抽象。

- *生产率高*

  由于封装了实现细节，DSL 优化在应用中编写或修改代码所需的工作量。

- *沟通畅快*

  DSL 可以帮助开发人员了解该领域，帮助领域专家验证实现是否符合要求。

然而，DSL 也有几个缺点。

- *DSL 很难开发*

  尽管编写 DSL 看起来很“酷”，但其开发成本不应该被低估。首先，实现 DSL 所需的技术并不简单（见下面的例子）。其次，良好的 DSL 比传统的 API 更难设计。后者往往遵循该语言 API 设计的惯用方法，保持 API 设计的一致性非常重要，这一点相对易于遵循。相反，由于每个 DSL 都是一门独一无二的语言，我们可以自由地创建习惯用法，以符合目标领域的独特特性。但自由度越大就越难找到最好的抽象。

• *DSL 很难维护*

  由于实现 DSL 所用的技术并不简单，随着对应领域的变化，DSL 可能需要更多长期的维护工作。为了更好的用户体验，这往往会牺牲实现的简洁性。

不过，如果经常使用 DSL 的话，设计良好的 DSL 可以成为构建可伸缩的、健壮的应用程序的有力工具。

从实现上看，DSL 可以分为内部和外部两种。

和 Scala 一样，内部 DSL（或嵌入型 DSL）是一门通用的编程语言。这里不需要特定用途的分析器。与此相反，外部 DSL 则是拥有自身语法和分析器的自定义语言。

内部 DSL 创建起来更容易，因为它们不需要专用的分析器。但另一方面，底层语言也约束了对特定领域的概念的表达。外部 DSL 没有这样的约束。你可以按任何的方式设计语言，只要你能编写出一个可靠的分析器就行。使用自定义的解析器颇具挑战性。其中为用户返回易懂的错误信息一直是解析器开发者面临的一个挑战。

## 20.1　DSL实例：Scala中XML和JSON DSL

十年前，XML 是互联网上机器与机器交流的通用语言。最近 JSON 已经取代了这个角色。Scala 对 XML 的支持部分以库的形式实现，另外也有一些内置语法上的支持。为了简化语言，并使其更容易使用第三方库来代替，这两者都在慢慢地废弃中。在 Scala 2.11 中，Scala 对 XML 的支持将从其余的库中提取出来，转而作为一个单独的模块。（见 Scaladoc，http://www.scala-lang.org/api/current/scala-xml/index.html#package。）我们建构的 `sbt` 中包含了这个模块，因此我们可以在本节中使用这一模块。

下面我们在 Scala 中简单地使用 XML，来查看 Scala 实现的这个 DSL 到底如何。这里接触的主要类型是 `scala.xml.Elem`（http://www.scala-lang.org/api/current/scala-xml/index.html#scala.xml.Elem） 和 `scala.xml.Node`（http://www.scala-lang.org/api/current/scala-xml/index.html#scala.xml.Node）：

```
// src/main/scala/progscala2/dsls/xml/reading.sc
import scala.xml._                                                    // ❶

val xmlAsString = "<sammich>...</sammich>"                            // ❷
val xml1 = XML.loadString(xmlAsString)

val xml2 =                                                            // ❸
<sammich>
  <bread>wheat</bread>
  <meat>salami</meat>
  <condiments>
    <condiment expired="true">mayo</condiment>
    <condiment expired="false">mustard</condiment>
  </condiments>
</sammich>

for {                                                                 // ❹
  condiment <- (xml2 \\ "condiment")
```

```
  if (condiment \ "@expired").text == "true"
} println(s"the ${condiment.text} has expired!")

def isExpired(condiment: Node): String =                    // ❺
  condiment.attribute("expired") match {
    case Some(Nil) | None => "unknown!"
    case Some(nodes) => nodes.head.text
  }

xml2 match {                                                // ❻
  case <sammich>{ingredients @ _*}</sammich> => {
    for {
      condiments @ <condiments>{_*}</condiments> <- ingredients
      cond <- condiments \ "condiment"
    } println(s"  condiment: ${cond.text} is expired? ${isExpired(cond)}")
  }
}
```

❶ 从 `scala.xml` 包（http://www.scala-lang.org/api/current/scala-xml/scala/xml/package.html）中导入公共 API 的声明。

❷ 定义一个包含 XML 的字符串，将其解析到 `scala.xml.Elem` 中，`XML` 对象（http://www.scala-lang.org/api/current/scala-xml/index.html#scala.xml.XML$）可以从很多源中读取 XML，也可以从 URL 中读取。

❸ 用 XML 字面量来定义 `scala.xml.Elem`。

❹ 在 XML 中遍历并提取字段。`xml \ "foo"` 只能匹配子节点，而 `\\ "foo"` 可以在必要的时候，深入遍历节点树。支持 XPath 表达式（http://www.w3.org/TR/xpath20），如表达式 `@expired` 表示寻找名为 `expired` 的属性。

❺ 辅助方法，用于寻找 `condiment` 节点中所有的 `expired` 属性。如果得到的是空序列或 `None`，方法返回 unknown!，否则提取序列中的第一个元素，并返回其文本值，该值应该是 true 或者 false。

❻ 用 XML 字面量做模式匹配，该表达式提取 `ingredients` 和一系列 `condiment` 标签，最后打印出提取的各个 `condiment` 中的数据。

`XML` 对象支持将 XML 保存到文件或保存到 `java.io.Writer`（http://docs.oracle.com/javase/8/docs/api/java/io/Writer.html）的几种方式：

```
// src/main/scala/progscala2/dsls/xml/writing.sc

XML.save("sammich.xml", xml2, "UTF-8")                     //❶
```

❶ 从 `xml2` 节点开始，将 XML 写入到文件 `sammich.xml` 中，文件编码采用 UTF-8。

Scala 在解析组合子库中，对 JSON 做了有限的支持，我们将在 20.3 节中进行探讨。现在有很多优秀的 Scala 及 Java 的 JSON 库，所以只有在需求有限时才考虑使用内置的 JSON 库。那么，你应该使用哪一种呢？如果你已经选择使用了某个大的框架，那就咨询其文档中推荐的库。否则，由于情况改变得非常快，你最好选择通过搜索寻找最适合的库。

# 20.2 内部DSL

Scala 语法的一些特性支持创建内部（嵌入型）DSL。

- *灵活的命名规则*

  由于你可以在命名时使用几乎所有的字符，所以很容易创建适合特定领域的名字，如同用代数符号表示具有相应特性的类型一样。例如，如果存在 `Matrix` 类型，你可以用 `*` 方法实现矩阵的乘法。

- *中缀和后缀符号*

  如果不能使用中缀表示法，使用 `*` 方法就意义不大。例如 `matrix1 * matrix2`。后缀表示法也很有用，如 `1 minute`。

- *隐含参数和默认参数值*

  这两种功能都可以减少样板代码，并隐藏复杂的细节。比如，DSL 中的每个方法都要传入上下文信息，该上下文信息就可以用隐含的值来代替。回想一下，许多 `Future` 方法（http://www.scala-lang.org/api/current/#scala.concurrent.Future）都有隐含的 `ExecutionContext` 参数（http://www.scala-lang.org/api/current/#scala.concurrent.ExecutionContext）。

- *类型类*

  隐式转换还可以为已经存在的类型“增加”方法。例如：`scala.concurrent.duration` 包（http://www.scala-lang.org/api/current/#scala.concurrent.duration.package）有对数字的隐式转换，允许编写 `1.25 minutes`，它将返回一个等于 75 秒的 `FiniteDuration` 实例（http://www.scala-lang.org/api/current/#scala.concurrent.duration.FiniteDuration）。

- *动态方法调用*

  正如我们在第 19 章所讨论的，`Dynamic` 特征（http://www.scala-lang.org/api/current/#scala.Dynamic）使得对象可以接受对任何方法或字段的调用，即使该类型不具有该名称所定义的方法或字段。

- *高阶函数和按名参数*

  两者均使得自定义的 DSL 看起来像本语言里的控制结构，这与我们在 3.10 节中看到的 `continue` 示例一样。

- *自类型标记*

  如果封闭范围内的实例中存在针对 DSL 实现中嵌套部分的自类型标记，DSL 实现中的嵌套部分便可以引用封闭范围内的实例。这一特性可以用来更新封闭范围内的状态对象。

- *宏*

  在一些高级的场景中可以使用新的宏，这一点我们将在第 24 章中学习。

下面我们来创建一个内部 DSL，用于计算每一期（两周）员工的工资单。DSL 将从总工资扣除一些项目，如税收、保险费、退休基金等，从而算出净工资。

一开始，我们先实现一些将在内部和外部 DSL 中都用的到的常见类型：

```
// src/main/scala/progscala2/dsls/payroll/common.scala
package progscala2.dsls.payroll

object common {
  sealed trait Amount { def amount: Double }                        // ❶

  case class Percentage(amount: Double) extends Amount {
    override def toString = s"$amount%"
  }

  case class Dollars(amount: Double) extends Amount {
    override def toString = s"$$$amount"
  }

  implicit class Units(amount: Double) {                            // ❷
    def percent = Percentage(amount)
    def dollars = Dollars(amount)
  }

  case class Deduction(name: String, amount: Amount) {              // ❸
    override def toString = s"$name: $amount"
  }

  case class Deductions(                                            // ❹
    name: String,
    divisorFromAnnualPay: Double = 1.0,
    var deductions: Vector[Deduction] = Vector.empty) {

    def gross(annualSalary: Double): Double =                       // ❺
      annualSalary / divisorFromAnnualPay

    def net(annualSalary: Double): Double = {
      val g = gross(annualSalary)
      (deductions foldLeft g) {
        case (total, Deduction(deduction, amount)) => amount match {
          case Percentage(value) => total - (g * value / 100.0)
          case Dollars(value) => total - value
        }
      }
    }

    override def toString =                                         // ❻
      s"$name Deductions:" + deductions.mkString("\n  ", "\n  ", "")
  }
}
```

❶ 定义一个封闭的继承结构，封装应该扣除的工资数额，扣除的项目要么是总额的固定百分比，要么是一个固定值。

❷ implicit 类，用于完成 `Double` 到对应正确 `Amount` 子类的转换。只用于内部 DSL。

❸ 定义一个类型表示扣除的项目，具有名称 `name` 和数额 `amount` 字段。

❹ 定义一个类型，表示所有的扣除项目。包含名称（如 Biweekly）和一个“除数”，该除数用于从年薪中计算本期的工资。

❺ 一旦 deduction 实例构造完成，就返回本期的总工资和净工资。

❻ 一般都会覆写 `toString` 方法，以返回我们想要的格式化形式。

以下是内部 DSL 的开头部分，其中的 `main` 函数显示了 DSL 使用的语法：

```
// src/main/scala/progscala2/dsls/payroll/internal/dsl.scala
package progscala2.dsls.payroll.internal
import scala.language.postfixOps                                   // ❶
import progscala2.dsls.payroll.common._

object Payroll {                                                   // ❷

  import dsl._                                                     // ❸

  def main(args: Array[String]) = {
    val biweeklyDeductions = biweekly { deduct =>                  // ❹
      deduct federal_tax          (25.0  percent)
      deduct state_tax            (5.0   percent)
      deduct insurance_premiums   (500.0 dollars)
      deduct retirement_savings   (10.0  percent)
    }

    println(biweeklyDeductions)                                    // ❺
    val annualGross = 100000.0
    val gross = biweeklyDeductions.gross(annualGross)
    val net   = biweeklyDeductions.net(annualGross)
    print(f"Biweekly pay (annual: $$${annualGross}%.2f): ")
    println(f"Gross: $$${gross}%.2f, Net: $$${net}%.2f")
  }
}
```

❶ 我们需要使用后缀表达式，如 `20.0 dollars`。

❷ 用来测试该 DSL 的对象。

❸ 导入这个 DSL，稍后会用到它。

❹ DSL 的实际使用。希望企业利益的相关者可以很容易地理解这里所表达的规则，甚至对其进行编辑。需要明确的是，这是 Scala 的语法。

❺ 将扣除打印处理，然后计算出这两周的净工资。

本程序的输出如下所示（`progscala2.dsls.payroll.internal.DSLSpec` 使用 ScalaCheck 进行更详细的验证）：

```
Biweekly Deductions:
  federal taxes: 25.0%
  state taxes: 5.0%
  insurance premiums: $500.0
  retirement savings: 10.0%
Biweekly pay (annual: $100000.00): Gross: $3846.15, Net: $1807.69
```

现在我们来看它是如何实现的：

```
object dsl {                                                    // ❶

  def biweekly(f: DeductionsBuilder => Deductions) =            // ❷
    f(new DeductionsBuilder("Biweekly", 26.0))

  class DeductionsBuilder(                                      // ❸
    name: String,
    divisor: Double = 1.0,
    deducts: Vector[Deduction] = Vector.empty) extends Deductions(
      name, divisor, deducts) {

    def federal_tax(amount: Amount): DeductionsBuilder = {      // ❹
      deductions = deductions :+ Deduction("federal taxes", amount)
      this
    }

    def state_tax(amount: Amount): DeductionsBuilder = {
      deductions = deductions :+ Deduction("state taxes", amount)
      this
    }

    def insurance_premiums(amount: Amount): DeductionsBuilder = {
      deductions = deductions :+ Deduction("insurance premiums", amount)
      this
    }

    def retirement_savings(amount: Amount): DeductionsBuilder = {
      deductions = deductions :+ Deduction("retirement savings", amount)
      this
    }
  }
}
```

❶ 将 DSL 的各个片段包装在一个对象中。

❷ `biweekly` 方法是定义扣除项目的入口。它构造了一个空的 `DeductionsBuilder` 对象，该对象会被原地修改（这是最简单的设计选择），以添加新的 `Deduction` 实例。

❸ 构建 `Deduction`，这里为了方便，继承了它。最终用户只能看到 `Deduction` 对象，但构建者有多个用于序列化表达式的额外方法。

❹ 支持 4 种扣除项目，这是第一个。注意它是如何原地更新 `Deduction` 实例的。

该 DSL 向上文所写的一样正常工作。但我认为，这远远不够完善。以下是存在的一些问题。

- *严重依赖 Scala 语法技巧*

  它利用中缀表示法、函数字面量等来开发 DSL，但用户很容易通过添加句号、括号和其他看起来无害的修改，而破坏代码。

- *语法约定非常随意*

  括号和花括号为什么这么放？为什么在匿名函数里需要 `deduct` 参数呢？

• *粗略的错误提示*

  如果用户输入了无效的语法，会抛出 Scala 的错误信息，而不是针对该领域特有的错误信息。

• *DSL 未能阻止用户做错误的事情*

  理想情况下，DSL 不会让用户在错误的情况下触发任何构造行为。而在这里，太多的构造行为是在 DSL 对象中可见的。没有什么可以阻止用户打乱正确的调用顺序，构造出实现代码中使用的内部类型的实例（如 `Percentage`）等等。

• *使用可变的实例*

  除非你是一个完美主义者，这也许并没有那么糟糕。像这样的 DSL 既不是为高性能而设计，也不会在多线程环境中运行。接受可变性可以用不多的妥协来简化实现。

这些问题的大部分都可以通过更多的努力来修复。

我最喜欢的内部 DSL 的例子是流行的 Scala 测试库、ScalaTest（http://scalatest.org）、Specs2（http://etorreborre.github.io/specs2）和 ScalaCheck（http://scalacheck.org）。它们提供了很好的面向开发者的 DSL 示例，使得创建 DSL 的努力物有所值。

# 20.3 包含解析组合子的外部DSL

给外部 DSL 编写解析器时，你可以使用诸如 Antlr（http://www.antlr.org）这样的解析器生成工具。不过，Scala 库本身也包含了解析组合子的库，可以用来解析大部分采用与上下文无关文法的外部 DSL。这个库定义内部 DSL 的方式十分特殊、引人注意，该方式使得解析器的定义与常见的语法标记特别像，就像扩展的巴科斯范式（EBNF）。

Scala 2.11 将解析组合子分离为一个单独的 JAR 文件，所以它现在是可选的。也有其他的第三方库，可以提供比这个库更好的性能，如 Parboiled 2（https://github.com/sirthias/parboiled2）。我们将使用 Scala 的库作为示例，其他库提供的 DSL 与此类似。

我们已经在 `sbt` 编译依赖中包含了解析器组合子的库（参见 Scaladoc，http://www.scala-lang.org/api/current/scala-parser-combinators/index.html#package）。

## 20.3.1 关于解析组合子

我们已经知道集合组合子可以用来构造数据转换器。类似地，解析器组合子则用来构造块解析器。解析器可以处理输入的特定位，如浮点数、整数等等，这些解析器又被组合在一起，以解析大的表达式。一个良好的解析器库应该支持序列和交替的表达式、重复、可选的表达式等等。

## 20.3.2 计算工资单的外部DSL

我们将重用前面的例子，但会采用更简单的语法。因为我们的外部 DSL 可以不必符合 Scala 语法。其他的一些修改使得解析器的构造更加容易，比如，在每个工资扣除项目之间添加逗号。

像之前一样，我们先给出 import 语句和主函数：

```
// src/main/scala/progscala2/dsls/payroll/parsercomb/dsl.scala
package progscala2.dsls.payroll.parsercomb
import scala.util.parsing.combinator._
import progscala2.dsls.payroll.common._                          // ❶

object Payroll {

  import dsl.PayrollParser                                       // ❷

  def main(args: Array[String]) = {
    val input = """biweekly {                                    // ❸
      federal tax          20.0  percent,
      state tax            3.0   percent,
      insurance premiums   250.0 dollars,
      retirement savings   15.0  percent
    }"""
    val parser = new PayrollParser                               // ❹
    val biweeklyDeductions = parser.parseAll(parser.biweekly, input).get

    println(biweeklyDeductions)                                  // ❺
    val annualGross = 100000.0
    val gross = biweeklyDeductions.gross(annualGross)
    val net   = biweeklyDeductions.net(annualGross)
    print(f"Biweekly pay (annual: $$${annualGross}%.2f): ")
    println(f"Gross: $$${gross}%.2f, Net: $$${net}%.2f")
  }
}
```

❶ 再次使用这里定义的部分公共类型。

❷ 表示工资扣除的“根”解析器。

❸ 输入。注意到，这次输入的是一个 `String`，而不是 Scala 表达式。

❹ 创建一个解析器实例，通过调用 `biweekly` 使用这个解析器，`biweekly` 返回用于整个 DSL 的解析器。`parseAll` 方法返回 `Parsers.ParseResult`（http://www.scala-lang.org/api/current/scala-parser-combinators/#scala.util.parsing.combinator.Parsers$ParseResult），我们再调用 `get` 得到 `Deductions`。

❺ 像上个示例一样，打印出输出。扣除的数量与上次不同，所以净工资也会不一样。

以下是解析器的定义：

```
object dsl {

  class PayrollParser extends JavaTokenParsers {                 // ❶

    /** @return Parser[(Deductions)] */
    def biweekly = "biweekly" ~> "{" ~> deductions <~ "}" ^^ { ds => // ❷
      Deductions("Biweekly", 26.0, ds)
    }

    /** @return Parser[Vector[Deduction]] */
    def deductions = repsep(deduction, ",") ^^ { ds =>           // ❸
```

```
    ds.foldLeft(Vector.empty[Deduction]) (_ :+ _)
  }

  /** @return Parser[Deduction] */
  def deduction = federal_tax | state_tax | insurance | retirement // ❹

  /** @return Parser[Deduction] */
  def federal_tax = parseDeduction("federal", "tax")              // ❺
  def state_tax   = parseDeduction("state", "tax")
  def insurance   = parseDeduction("insurance", "premiums")
  def retirement  = parseDeduction("retirement", "savings")

  private def parseDeduction(word1: String, word2: String) =      // ❻
    word1 ~> word2 ~> amount ^^ {
      amount => Deduction(s"${word1} ${word2}", amount)
    }

  /** @return Parser[Amount] */
  def amount = dollars | percentage                               // ❼

  /** @return Parser[Dollars] */
  def dollars = doubleNumber <~ "dollars" ^^ { d => Dollars(d) }

  /** @return Parser[Percentage] */
  def percentage = doubleNumber <~ "percent" ^^ { d => Percentage(d) }

  def doubleNumber = floatingPointNumber ^^ (_.toDouble)
 }
}
```

❶ 类通过其中的方法定义了语法和解析器。

❷ 顶层解析器，该解析器通过组合小解析器创建得到。入口方法 `biweekly` 返回 `Parser[Deductions]`，这是一个可以从字符串中解析出工资扣除项目的解析器，它返回一个 `Deductions` 对象。我们稍后会讨论它的语法。

❸ 解析一组由逗号分隔的扣除项目。其中逗号的添加简化了解析器的实现。`repsep` 方法可以解析任意个数的扣除项目表达式。

❹ 4 个扣除项目。

❺ 调用辅助函数，构造解析这 4 个项目的解析器。

❻ 用于处理这 4 个扣除项目的辅助方法。

❼ 解析 amount，amount 是由 `dollars` 和 `percent` 组成的。同时创建了对应的 `Amount` 实例。

下面我们仔细观看 `biweekly` 的写法，为方便讨论我们再一次给出 `biweekly` 的写法：

```
"biweekly" ~> "{" ~> deductions <~ "}"                         // ❶
  ^^ { ds => Deductions("Biweekly", 26.0, ds) }                // ❷
```

❶ 找到 3 个结束标记（terminal token），`biweekly`、`{`、`}`，以及对 `{…}` 中内容计算扣除的结果。类似箭头的操作符（其实是方法名）`~>` 和 `<~` 表示将 `~` 一侧的标记丢掉。于是语法标记都去掉了，只留下 `deductions`。

❷ ^^ 将左边（标记）和右边（语法规则）分开。语法规则带一个参数，是保留下来的标记。如果存在多个标记，则需要使用一个形如 { case t1 ~ t2 ~ t2 => ... } 的偏函数字面量。在这个例子中，ds 是 Deduction 实例的 Vector，用于构造 Deductions 实例。

需要注意的是，这里并不需要内部 DSL 中的 DeductionsBuilder。详尽的验证可以参阅 progscala2.dsls.payroll.parsercomb.DSLSpec 中的测试，该测试使用 ScalaCheck 进行验证。

## 20.4 内部DSL与外部DSL：最后的思考

下面我们来比较一下用户所写的内部 DSL 和外部 DSL。这里再次给出内部 DSL：

```
val biweeklyDeductions = biweekly { deduct =>
  deduct federal_tax          (25.0  percent)
  deduct state_tax            (5.0   percent)
  deduct insurance_premiums   (500.0 dollars)
  deduct retirement_savings   (10.0  percent)
}
```

下面给出外部 DSL：

```
val input = """biweekly {
  federal tax          20.0  percent,
  state tax            3.0   percent,
  insurance premiums   250.0 dollars,
  retirement savings   15.0  percent
}"""
```

外部 DSL 更简单，但用户必须将 DSL 语句放在字符串中。所以，外部 DSL 中不存在代码补全、纠错、语法高亮和其他 IDE 特性。

但另一方面，外部 DSL 更容易实现（也更有趣）。它对 Scala 解析技巧的依赖也相对较少。

你必须权衡其中的取舍，做出最适合你的选择。如果 DSL 与 Scala“足够接近”，在 Scala 内部花费合理的努力即可实现，并有不错的健壮性，那么内部 DSL 的用户体验通常会更好。显然这也是前面提到的测试库的最佳选择。如果 DSL 与 Scala 的语法相差太远，这也许是因为 DSL 是一门类似 SQL 的大众语言，使用带引号的字符串的外部 DSL 可能是最好的选择。

回顾 5.3.1 节，我们可以实现自己的字符串插值器。这样，我们就可以用稍微简单一点的语法来封装一个用组合子构建的解析器。例如，如果你实现了一个 SQL 语法分析器，用户便可以使用 sql"SELECT * FROM table WHERE ...;" 来调用该分析器，而不必像我们之前一样显式地调用该解析器 API。

## 20.5 本章回顾与下一章提要

人们在创建 DSL 时很容易放弃。创建 Scala 中的 DSL 可以说相当有趣，但是，要创建符合客户的可用性需求的健壮 DSL，其中的工作量不可低估。除了巨大的工作努力，还需要进行长期的维护与支持。

在下一章中，我们将探讨 Scala 的工具和库。

# 第 21 章

# Scala工具和库

本章将对我们曾经使用过的 Scala 工具进行详细地讲解，这些工具包括编译器 `scalac`、REPL 中的 `scala` 命令等。我们将讨论构造工具选项、IDE 以及如何在文本编辑器中集成这些工具，之后我们将学习 Scala 的测试库，最后我们将列举某些可能有用的第三方 Scala 库。

## 21.1　命令行工具

尽管你的大多数工作都是在 IDE 或 SBT REPL 中完成的，理解命令行工具的工作原因仍能为你带来一些额外的灵活度，而如果图形化工具无法正常工作，这些工具也能作为一个备用方案。在大多数时候，你都会在 SBT 构造文件或 IDE 设置中配置编译器标志，之后你会通过 `console` 命令，在当前 `SBT` 会话中运行 REPL。

在 1.2 节，我们讲解了如何安装命令行工具。所有命令行工具都位于 SCALA_HOME/bin 文件夹中，其中 SCALA_HOME 是 Scala 的安装路径。

你可以访问 http://www.scala-lang.org/documentation 页面，了解更多关于命令行工具的相关信息。

### 21.1.1　命令行工具：scalac

`scalac` 命令会编译 Scala 源文件，并生成 JVM 类文件。

`scalac` 命令只是一个封装了 `java` 命令的 shell 脚本而已，该脚本将包含 Scala 编译器 `Main` 对象的名字传递给了 `java` 命令。除此之外，该脚本还将 Scala 使用的 JAR 文件添加到 `CLASSPATH` 中，并定义了许多与 Scala 相关的系统参数。

scalac 命令运行格式如下：

```
scalac <options> <source files>
```

我们回顾下 1.3 节的内容，源文件名并不需要与文件中定义的 public 类的类名相吻合。实际上，你可以在同一个文件中定义多个 public 类。类似地，包名也无需与路径结构相一致。

但是，为了遵守 JVM 需求，Scala 会为每个类型生成一个独立的类文件，类文件名与类型名相同。同样，这些类文件会根据所在的包声明体的名称写入对应的文件夹中。

表 21-1 列举了 scalac 的选项。使用 2.11.2 版本的编译器时，执行 scalac -help 命令便能得到这些选项的说明（表中对这些选项说明做了少许修改）。

表21-1：scalac命令选项

| 选　　项 | 说　　明 |
|---|---|
| -Dproperty=value | 将 -Dproperty=value 选项直接传递给运行时系统 |
| -J*flag* | 将 Java 标志直接传递给运行时系统 |
| -P*plugin:opt* | 将选项 opt 传递给某一编译器插件 |
| -X | 打印 scalac 高级选项的摘要信息（表 21-3 将对这些高级选项进行了讲解） |
| -bootclasspath *path* | 覆写用于启动的类文件的路径 |
| -classpath *path* | 设置用户类文件的查找路径 |
| -d *directory or jar* | 指定生成类文件的地址 |
| -dependencyfile *file* | 指定记载了依赖关系的文件 |
| -deprecation | 假如用户使用了已过时的 API，输出信息中将不再打印警告信息和这些 API 对应的源文件位置 |
| -encoding *encoding* | 指定源文件使用的字符编码 |
| -explaintypes | 出错时更详细地解释类型错误 |
| -extdirs *dirs* | 重新设置已安装编译器扩展的路径 |
| -feature | 假如没有显式地导入那些应显式导入的功能，将显示对应的警告信息和代码位置信息 |
| -g:*level* | 指定生成的调试信息的级别，包括 none、source、line、vars（默认级别）和 notailcalls |
| -help | 打印标准选项的摘要信息 |
| -javabootclasspath *path* | 对 Java 的启动 classpath 进行覆写 |
| -javaextdirs *path* | 覆写 Java 的 extdirs 的类路径 |
| -language:*feature* | 启动一个或多个语言功能，这些功能包括：dynamics、postfixOps、reflectiveCalls、implicitConversions、higherKinds、existentials、以及 experimental.macros（输入多个功能时，这些功能以逗号分割，不能包含空格） |
| -no-specialization | 忽略所有的 @specialize 标注 |
| -nobootcp | 不要对 Scala 语言的 JAR 包使用启动类路径 |
| -nowarn | 不生成警告信息 |
| -optimise | 通过对程序进行优化，生成运行速度更快的字节码 |
| -print | 不考虑任何 scala 相关的功能，仅仅打印程序内容 |
| -sourcepath *path* | 指定源文件的查找路径 |

（续）

| 选　　项 | 说　　明 |
|---|---|
| -target:*target* | 指定运行目标文件的目标平台，如：jvm-1.5（不推荐）、jvm-1.6（默认）、jvm-1.7 |
| -toolcp *path* | 将指定目录添加到运行类路径中 |
| -unchecked | 有时候 scala 需要根据猜测，才能决定生成的代码。此时如果开启了这一选项，scalac 便会打印出一些额外的警告信息 |
| -uniqid | 在调试的输出信息中，给所有的标识符添加一个唯一的标签 |
| -usejavacp | 使用 java.class.path 路径作为 classpath |
| -usemanifestcp | 使用 manifest 文件中定义的 classpath |
| -verbose | 在输出中打印编译器正在执行的工作信息 |
| -version | 打印产品版本信息，之后退出 |
| @*file* | 指定了一个文本文件，该文件中记录了编译器参数（编译器选项和需要编译的文件） |

接下来，我们将选择一部分选项进行讨论。

假如在变量名字或允许的符号中使用非 ASCII 的字符（例如，使用 Unicode\u21D2 而不是 =&tg 字符来表示 => 符号），你需要使用 -encoding UTF8 选项。

如果出现类型错误时你希望能够得到更完整的解释信息，请使用 -explaintypes 选项。

从 Scala 2.10 开始，一些更高级的语言功能都被设置为可选功能，因此开发团队可以选择启动他们希望使用的功能。这也是解决 Scala 复杂性问题工作的一部分，如果需要的话，我们仍然可以使用 Scala 的高级结构。使用 -feature 选项后，假如源代码未显式引入这些可选功能，并且未开启 -language:< 功能名 > 编译器标记，那么 scala 便会发出警告信息，并列出使用了这些功能的所有的源码位置。

scala.language 对象（http://www.scala-lang.org/api/current/#scala.language$）定义了一组可选语言 trait，相关的 Scaladoc（http://www.scala-lang.org/api/current/#scala.language$）页面也解释了这些功能被设置为可选功能的原因。下面的表 21-2 列出了这些功能。

表21-2：可选的语言功能

| 功能名 | 功能描述 |
|---|---|
| dynamics | 启用 Dynamic 特征（请参照第 19 章的相关内容） |
| postfixOps | 启用后缀操作符（例如，100 toString） |
| reflectiveCalls | 允许用户使用结构化类型（请参考 24.2.1 节的相关内容） |
| implicitConversions | 允许用户定义隐式方法及成员（请参考 5.3 节的相关内容） |
| higherKinds | 允许用户编写 higher-kinded 类型（请参考 15.5 节的相关内容） |
| existentials | 允许用户编写存在类型（existential types，请参考 14.9 节的相关内容） |
| experimental | 包含了一些较新的功能，这些功能目前尚未在生产环境中测试过。目前只有宏是 Scala 语言中唯一的试验功能（请参考 24.4 节的相关内容） |

-X 高级选项（即 scalac -X 和 scala -X）提供了控制是否输出复杂的诊断信息、调整编译

器行为、控制使用的试验性扩展和插件等功能。表 21-3 对其中的某些选项进行了说明。

表 21-3：-X的一些高级选项

| 选项名 | 描　述 |
|---|---|
| `-Xcheckinit` | 对字段访问器进行封装，假如访问了未初始化的字段，便会抛出异常（请参考 11.4 节的相关内容） |
| `-Xdisable-assertions` | 不生成断言和假设信息 |
| `-Xexperimental` | 启用某些试验性的扩展功能 |
| `-Xfatal-warnings` | 只要存在任何警告信息，编译过程便以失败告终 |
| `-Xfuture` | 启用 future 语言特性 (if any for a particular release) |
| `-Xlint` | 启用 `lint` 推荐的额外警告信息 |
| `-Xlog-implicit-conversions` | 每出现一次隐式转换，便打印出一条日志信息 |
| `-Xlog-implicits` | 如果出现了不可用的隐式，便会详细地显示不适用的原因 |
| `-Xmain-class` *path* | 指定 JAR 文件 manifest 中的主类入口，只有使用 `-d jar` 选项才起作用。 |
| `-Xmigration:`*v* | 假如某些结构的行为从 Scala 的第 v 版起发生了变化，那么将给出警告信息 |
| `-Xscript` *object* | 将源代码文件视为脚本，并将文件内容封装到 main 方法内 |
| `-Y` | 打印关于私有选项的概要信息。所谓私有选项，是用于实现新的语言特性的那些选项 |

在下面的示例中，我们第一次在 REPL 中使用了 `-Xlint` 选项。从该示例中，你能观察到 `-Xlint` 选项增加了一些额外的警告信息：

```
$ scala -Xlint
Welcome to Scala version 2.11.2 ...
...
scala>  def hello = println("hello!")
<console>:7: warning: side-effecting nullary methods are discouraged:
  suggest defining as `def hello()` instead
       def hello = println("hello!")
           ^
hello: Unit
```

所有返回 `Unit` 类型的函数都应该具有副作用。在这个示例中，我们会打印输出信息。依照 Scala 代码的规范，只有当函数不会有任何副作用时，才能将其声明为*零元方法*（nullary method），即不为该方法指定参数列表。由于 hello 方法具有副作用，因此此处出现了一条警告信息。

假如你在编译脚本文件时，希望将其作为普通 Scala 源代码文件进行处理，那么 `-Xscript` 选项便派上了用场。这样做的好处在于减少重复编译该脚本所导致的启动开启。

> 我推荐读者在日常工作中使用 `-deprecation`、`-unchecked`、`feature` 和 `-Xlint` 选项。（对于某些代码库而言，使用 `-Xlint` 选项也许会产生非常多的警告信息。）这些选项除了能够帮助我们减少 bug 之外，还能提醒我们不要使用过时库。与其他的标志一样，我们会在代码示例对应的 build.sbt 文件中使用这些标志。

## 21.1.2 Scala命令行工具

如果指定了待运行的程序，scala 命令会执行该程序。假如未指定程序名，scala 会启动 REPL。与 scalac 相似，scala 同样也是一个 shell 脚本。输入下列语句，便能运行 scala 命令。

```
scala <options> [<script|class|object|jar> <arguments>]
```

scalac 命令行选项同样适用于 scala 命令，除此之外，scala 还能接受表 21-4 列出的额外选项。

表21-4：额外的scala命令选项

| 选　项 | 描　述 |
|---|---|
| -howtorun | 执行的是什么类型的程序呢？脚本、对象、jar 包还是让 scala 自己猜测（默认行为） |
| -i *file* | 在启动 REPL 之前，预先加载文件内容 |
| -e *string* | 执行 string 字符串，将其视为输入到 REPL 上的命令 |
| -save | 将编译过的脚本存储成 JAR 文件，这样一来，之后使用该脚本时无需对其重编译 |
| -nc | 不运行编译守护进程。此时，离线编译器 fsc 将会自动运行，以避免每次重启编译器的开销 |

运行 scala 命令时，将会对第一个非选项的参数进行解释，并将其作为 scala 执行的程序。假如未指定该参数，那么将启动 REPL 终端。假如用户指定了需要执行的程序，那么出现在程序参数之后的所有参数都将作为 args 数组传递给程序。我们在之前的许多示例中都使用过 args 数组。

另外，除非指定了 -howtorun 选项，否则 scala 会自行推测程序的类型。假如传入了 Scala 源码的文件，那么 scala 会将其视为脚本，并执行该脚本。假如传入了一个定义了 main 方法的类文件或具有 Main-Class 属性的 JAR 文件，scala 便会运行一个典型的 Java 程序。

如果希望能在输入命令之前预加载某一文件，你可以在交互模式下使用 -i file 选项。即便进入了 REPL 模式，你还能使用 :load *filename* 命令加载文件。假如你开启了 REPL 模式，并且需要重复执行相同的命令，那么使用这样一个预加载文件会很有用。

使用 REPL 时，你可以处理许多条命令。输入 :help 命令可以查看这些命令的概要描述。表 21-5 列出了 Scala 2.11.2 版提供的可用命令，我们对其做了少量的编辑。

表 21-5：Scala REPL中可以使用的命令

| 命　令 | 描　述 |
|---|---|
| :cp path | 将某个 JAR 包或某个目录添加到 classpath 中 |
| :edit *id or line* | 编辑输入历史 |
| :help *[command]* | 打印概括性的帮助信息或某个命令相关的帮助信息 |
| :history *[num]* | 显示命令的历史纪录（*num* 是可选参数，代表了显示的命令条数） |
| :h? *string* | 查询历史纪录 |
| :imports*[name name …]* | 显示加载文件的历史记录，以便了解这些定义名称的来源 |

（续）

| 命　　令 | 描　　述 |
| --- | --- |
| :implicits [-v] | 显示作用域中定义的各类隐式（-v 是可选项，假如包含了 -v 选项，会显示更详细的输出内容） |
| :javap *path or class* | 对指定的文件或指定名称的类进行反编译 |
| :line *id or line* | 在历史信息末尾处放置几行内容 |
| :load *path* | path 指定了某一文件，scala 会对文件中的各行内容进行解释 |
| :paste [-raw]*[path]* | 进入粘贴模式，或者复制某个文件的内容 |
| :power | 进入超级用户模式（请查看输入该命令后打印的信息） |
| :quit | 退出 Scala 解释器（也可以使用 Ctrl-D 快捷键） |
| :replay | 重置执行，并重新运行之前输入的所有命令 |
| :reset | 将 REPL 重新设置为起始状态，并移除所有的会话入口 |
| :save *path* | 将该会话信息存储到文件中，之后可以使用该文件执行 replay 操作 |
| :sh *command line* | 执行一条 shell 命令（执行结果类型为 implicitly => List[String]） |
| :settings*[+ or -]options* | 启用 (+)/ 关闭 (-) 标志位，设置编译器选项 |
| :silent | 关闭 / 开启自动打印结果值的功能 |
| :type [-v]*expr* | 在不对表达式进行估值的情况下展示表达式类型 |
| :kind [-v]*expr* | 展示表达式类型的 kind 值 |
| :warnings | 找到最近一行具有警告信息的代码，并显示对应的警告信息 |

使用 :power 命令可以进入“超级用户模式”，该模式提供了额外的命令用于查看内存中的数据，如抽象语法树、解析器属性等。你还能对编译器进行操作。

假如要执行的脚本的使用频率较高，运行 scala 命令来执行这些脚本就会显得有些麻烦。在 Windows 和类 Unix 系统中，你可以编写独立的 Scala 脚本，无需通过 scalac 脚本文件名的方式执行该脚本。

对于类 Unix 系统，下面的示例说明了如何编写一个可执行脚本。请记住你需要为脚本文件设置可执行的权限，如执行 chmod +x secho 命令：

```
#!/bin/sh
# src/main/scala/progscala2/toolslibs/secho
exec scala "$0" "$@"
!#
print("You entered: ")
args.toList foreach { s => printf("%s ", s) }
println
```

下面列出了该脚本可能的应用场景：

```
$ secho Hello World
You entered: Hello World
```

Windows 平台的 .bat 命令脚本与之前的脚本较为相似，如下所示：

```
::#!
@echo off
```

```
call scala %0 %*
goto :eof
::!#
print("You entered: ")
args.toList foreach { s => printf("%s ", s) }
println
```

### scala命令与scalac命令的局限

使用 scala 命令运行源文件或是使用 scalac 编译源文件都具有一些局限性。

通过 scala 命令执行的脚本会被封装到一个匿名对象中，封装后的脚本与下面示例或多或少有些相似：

```
object Script {
  def main(args: Array[String]): Unit = {
    new AnyRef {
      // 你的脚本代码将被插入到此处。
    }
  }
}
```

Scala 对象中无法嵌入包声明体，这意味着你无法在脚本中声明包。这解释了本书中所有声明了包的示例都必须经过编译之后再运行的原因。

反过来，有些脚本无法使用 scalac 进行编译。除非你在编译时使用了 -Xscript *object* 选项，选项中的 option 表示被编译对象的名称。在之前的示例中，*object* 对应了对象名：Script。换言之，编译器选项 -Xscript 会对脚本内容进行封装，其封装器正是 REPL 隐式封装时所使用的封装器。

之所以编译某些脚本时需要使用对象封装器，是由于 Scala 不允许在类型外定义或调用函数。将 scala 命令作为脚本，下面的示例能够正常地运行：

```
// src/main/scala/progscala2/toolslibs/example.sc

case class Message(name: String)

def printMessage(msg: Message) = println(msg)

printMessage(new Message("This works fine with the REPL"))
```

不过，假如使用 scalac 编译该脚本，但未添加 -Xscript 选项时，你会发现系统产生了下列错误：

```
example.sc:3: error: expected class or object definition
def printMessage(msg: Message) = println(msg)
^
example.sc:5: error: expected class or object definition
printMessage(new Message("This works fine with the REPL"))
^
two errors found
```

我们应该使用下列方式编译并执行该脚本：

```
scalac -Xscript MessagePrinter src/main/scala/progscala2/toolslibs/example.sc
scala  -classpath . MessagePrinter
```

由于脚本会被放置在默认包内，因此生成的类文件将位于当前文件夹：

```
MessagePrinter$$anon$1$Message$.class
MessagePrinter$$anon$1$Message.class
MessagePrinter$$anon$1.class
MessagePrinter$.class
MessagePrinter.class
```

对每个文件都执行 `javap -private`（我们将在下一节中讲解该命令），便能查看这些文件所包含的声明体。`-p` 标志会提醒 javap 程序列出所有的成员，包括私有成员和受保护成员（运行 `javap -help`，可以查看所有的选项）。在 javap 命令中需要省略 `.class` 后缀，如 `javap MessagePrinter$$anon$1$Message$`。

`MessagePrinter` 和 `MessagePrinter$` 都是 `scalac` 命令生成的封装类，这两个封装类为脚本提供了“应用程序”入口。`MessagePrinter` 定义了我们所需的静态 `main` 方法。

`MessagePrinter$$anon$1` 是 scala 生成的一个 Java 类，该类封装了整个脚本。脚本中定义的 `printMessage` 方法变成了此类的一个私有方法。

`MessagePrinter$$anon$1$Message` 是 `Message` 类。

`MessagePrinter$$anon$1$Message$` 是 `Message` 的伴生对象。

## 21.1.3 scalap和javap命令行工具

假如你希望理解 Scala 结构体是如何使用 JVM 字节码实现的，反编译器能派上用场。反编译器能帮助你理解 Scala 名字为了能够转换成与 JVM 兼容的名字，遭受了怎样的破坏。

老资格的 `javap` 出自于 JDK。就像是输出 Java 源代码一样，即使是处理 scalac 编译 Scala 代码后产生的文件，`javap` 也能输出文件中包含的声明体。因此，假如你希望查看 Scala 中的定义体是如何被映射成有效的 Java 字节码，运行 `javap` 命令查看这些类文件是一个很好的方法。

Scala 提供了 `scalap` 工具，该工具将输出文件中包含的声明体，这些输出看上去就像是 Scala 源代码一样。不过，由于疏忽，Scala 2.11.0 版和 2.11.1 版没有将 `scalap` 包含在发行版本中。你可以访问 http://mvnrepository.com/artifact/org.scala-lang/scalap/2.11.1，下载 2.11.1 版的 `scalap` 的 JAR 文件，然后将 JAR 文件复制到安装的 lib 目录中。Scala 2.11.2 版包含 `scalap` 工具。

执行 `scalap -cp . MessagePrinter` 命令，对之前章节中出现的 MessagePrinter.class 进行反编译。如果一切正常的话，你将可以看到下列输出（我们对代码格式进行了修正，以符合书页的大小）：

```
object MessagePrinter extends scala.AnyRef {
  def this() = { /* 编译后的代码 */ }
  def main(args : scala.Array[scala.Predef.String]) : scala.Unit = {
```

```
    /* 编译后的代码 */
  }
}
```

我们将其与 javap -cp . MessagePrinter 命令的输出进行对比：

```
Compiled from "example.sc"
public final class MessagePrinter {
  public static void main(java.lang.String[]);
}
```

现在我们看到的 main 方法的声明体就和阅读 Java 源代码中的 main 声明体一样了。

这些工具均提供了 -help 命令选项，该选项描述了这些工具提供的功能选项。

下面进入练习环节，我们将对由 progscala2/toolslibs/Complex.scala 生成的类文件实施反编译，该文件实现了 Complex 这一数值类。我们之前已经使用 SBT 对 Complex.scala 文件进行了编辑。现在，我们将对产生的类文件运行 scalap 和 javap 命令。其中我们在源文件中声明了包 toolslibs，请留意我们是如何指定包名的。使用下列 javap 命令，我们便能对 Scala 2.11 构建出的类文件实施反编译：

```
javap -cp target/scala-2.11/classes toolslibs.Complex
```

+ 方法与 - 方法在类文件中是如何编码的呢？那些存在或假想的字段的 getter 方法的名字是什么呢？ 这些字段对应的 Java 类型又会是什么呢？ scalap 和 javap 会输出什么内容呢？

### 21.1.4　scaladoc命令行工具

scaladoc 命令与 javadoc 命令相似，该工具会基于 Scala 源代码生成文档。这些文档被称为 Scaladoc。与 javadoc 一样，scaladoc 解析器也支持相同的 @ 注释，如 @author、@param 等。

在项目中使用 scaladoc 的最简单方法是执行 SBT 的 doc 任务。

### 21.1.5　fsc命令行工具

fsc 是快速 scala 编译器（fast scala compiler）的简写，为了能够更快地启动编译器，fsc 会以 daemon 进程的方式运行，以便大幅消除编译器的启动开销。假如你需要重复的执行脚本（举个例子，在 bug 能够复现之前，你也许需要重新运行一组测试集），那么使用 fsc 便尤为有用。实际上，scala 命令会自动运行 fsc 工具，但你也可以直接调用该工具。

## 21.2　构建工具

大多数新项目都使用 SBT（http://www.scala-sbt.org/）构建工具，因此我们将做集中讨论。不过目前已经存在一些基于其他类型的构造工具的 Scala 插件，这其中包括 Ant（http://ant.apache.org/）、Maven（mvn）（http://maven.apache.org/）和 Gradle（http://www.gradle.org/）。

### 21.2.1 SBT：Scala标准构建工具

SBT 是一款可用于构造 Scala 项目和 Java 项目的复杂工具。它提供了很多的配置选项以及插件功能。我们在所有的代码示例中都使用了这一工具。下面我们将对 SBT 的功能进行稍微更加深入的了解，其中便会涉及 SBT 实际的代码构造文件的结构。不过，我们对这些知识也只能蜻蜓点水，不会涉及太深的内容。（如果想了解更多的信息，请参考 SBT 的相关文档（http://www.scala-sbt.org/documentation.html）；也可以参考由 Joshua Suereth 和 Matthew Farwell 合作编写的《SBT 实战》一书。）

现在，你已经安装好了 SBT 工具。假如希望进行基于 JVM 的 web 开发，也请你查阅一下新的 `sbt-web` 项目（https://github.com/sbt/sbt-web）。该项目增加了可用于构建并管理 web 资源的插件，其中的 web 资源包括 HTML 网页和根据模版语言而生成的 CSS 文件等。

> 最快上手 SBT 的方法是复制现有的构造文件并对其进行修改。例如，你可以参考 Activator 模版（http://www.typesafe.com/activator/templates）中的这些构造文件。

SBT 与 Maven 相似，它们都内嵌了我们所需要执行的一些任务，如编译、自动化测试等；与此同时，它们还定义了任务之间的依赖关系，例如：编译应在测试之前发生。SBT 的构造文件中定义了项目的元数据，例如：项目名、发布版本等信息。此外，构造文件中还使用 Maven 规范和 Maven 库定义的各个组件之间的依赖关系（不过在解析依赖关系时，应用了 Ivy 工具（http://ant.apache.org/ivy）），并定义了一些自定义的数据。其中构造文件中使用了基于 Scala 的一门 DSL 语言。

SBT 的构建活动由一个或多个构建文件所定义，构建文件的数量取决于项目的复杂度和定制化要求。对于本书的示例而言，尽管同时支持两个版本的 Scala 会增加些许复杂度，但这些示例项目还都相对简单。

主构建文件位于代码示例的顶层文件夹中，其名为 `build.sbt`。在 project 子目录里有两个额外的文件：build.properties 文件，它定义了我们希望使用的 SBT 的版本信息；另一个文件 plugins.sbt 则会在生成 Eclipse 项目文件时为该项目添加 SBT 的相关插件。（IntelliJ IDEA 可以直接导入 SBT 项目。）将 build.sbt 文件放在 project 目录中也很常见。

下面，我们将查看一个简化版的 build.sbt 文件，该文件的头部包含了一些定义体。

```
name := "Programming Scala, Second Edition: Code examples"

version := "2.0"

organization := "org.programming-scala"

scalaVersion := "2.11.2"
```

构造文件中定义了变量，变量定义体采用了类似于 `name := "Programming Scala, ..."` 这样的格式。为了能更容易地定位到定义体的结束位置，目前构造文件的 DSL 要求每个定义之间包含一个空行。假如忘记了在定义体之间添加空行，你将会看到一条与之相关的错误信息。

下面列出了文件中定义的依赖关系（一些依赖关系已被删减）：

```
libraryDependencies ++= Seq(
  "com.typesafe.akka"      %% "akka-actor"      % "2.3.4",
  "org.scalatest"          %% "scalatest"       % "2.2.1"  % "test",
  "org.scalacheck"         %% "scalacheck"      % "1.11.5" % "test",
  ...
)
```

有的时候，我们需要使用一个序列 Seq 对象用于定义变量。例如，我们依赖了一组库，包括版本号为 2.3.4 的 Akka actor 库、ScalaTest 库、ScalaCheck 库（请参见 21.4 节），此处还省略了一些其他的依赖库。

我们并没有列出 Maven 兼容的仓库定义。这些仓库记录了查找依赖库的 Internet 地址。SBT 已经知道了一些标准的仓库位置，不过你仍然可以自定义一些仓库。你能在 project/plugins.sbt 文件中找到指定 Maven 仓库的相关示例。关于如何指定仓库，请参考 SBT 中解析器 resolver 的定义。

libraryDependencies 剩余的部分内容定义了一些实际中可能会用到的定义，我们也会在这里列出该部分内容。

最后，我们为 scalac 和 javac 命令指定了所需的编译器标示：

```
scalacOptions = Seq(
  "-encoding", "UTF-8", "-optimise",
  "-deprecation", "-unchecked", "-feature", "-Xlint", "-Ywarn-infer-any")

javacOptions  ++= Seq("-Xlint:unchecked", "-Xlint:deprecation")
```

我们可以在构造文件中定义 REPL 启动控制台后自动执行的一些语句。尽管我们未使用这一功能，不过了解该功能还是很有帮助的。请参考下面的示例：

```
initialCommands in console := """
  |import foo.bar._
  |import foo.bar.baz._
  |""".stripMargin
```

这些选项与之前讨论的 scala 命令中的 -i file 选项类似。控制台提供了两个额外的变量。第一个变量是 consoleQuick 变量（你也可以写成 console-quick），该变量并不会优先对你的代码进行编译。假如代码当前并没有执行构建操作（或者构建操作需要花费很长时间），而你又希望尝试执行某些代码，那么 consoleQuick 变量便能派上用场。

另一个变量是 consoleProject（也可以写作 console-project），该变量会忽略你所编写的代码，但却会加载 SBT 定义的资源、CLASSPATH 中定义的一些构建资源以及一些有用的导入文件。

控制台中的 initialCommands 声明同样适用于 consoleQuick 变量，而且你还能为 consoleQuick 单独定制 initialCommands。相比之下，控制台中的 initialCommands 无法用于 consoleProject 变量，不过你能为 consoleProject 变量单独定制 initialCommands 值：

```
initialCommands in console := """println("Hello from console")"""
initialCommands in consoleQuick := """println("Hello from consoleQuick")"""
```

```
initialCommands in consoleProject := """println("Hello from consoleProject")"""
```

你还可以使用对应的 `cleanup` 命令，该命令能够帮助你对可能经常使用的资源进行自动清理，如数据库会话资源。

### 21.2.2 其他构建工具

由于其他构建工具插件均使用了 SBT 工具所使用的相同的增量编译技术，因此无论选用何种构建工具，构建时间大致相同。

你可以在 Scala 发布版中找到 lib/scala-compiler.jar 文件，该文件中定义了 `scalac`、`fsc` 以及 `scaladoc` 对应的 Ant 任务。Scala 中的 Ant 任务与对应 Java 中的非常类似。执行 Ant 时需要使用配置文件 build.xml，其中“Scala Ant 任务”页面（http://www.scala-lang.org/old/node/98）对该配置文件进行了讲解。尽管这个页面已经有些年头，但目前仍然有效。

我们可以在 GitHub（http://davidb.github.io/scala-maven-plugin/）中找到适用于 Scala 语言的 Maven 插件。该插件会帮你自动下载 Scala 语言，因此使用时无需安装 Scala。

除此之外，我们还可以使用 Eclipse 和 IntelliJ 中集成的 Maven 环境。

假如更青睐 Gradle，你可以在 Gradle 的相关网站中（http://www.gradle.org/docs/current/userguide/scala_plugin.html）找到 Gradle 插件的详细信息。

## 21.3 与IDE或文本编辑器集成

如果你具备 java 开发的背景，你也许会被 Java IDE 提供的丰富功能宠坏。自本书第一版发布以来，Scala IDE 插件已经得到了很大的发展，也有一些专业的团队专门负责这些工具。尽管对 Scala 的工具支持还是没有 Java 工具支持那样的成熟，但是所有必需的工具也都已经存在了。大多数 IDE 中的 Scala 插件都集成了 SBT 或 Maven 构造工具，并提供了语法高亮、部分自动化重构功能以及新的 worksheet 功能。worksheet 功能可以很好地取代命令行中的 REPL 环境。

假如你使用 Eclipse，请参考“Scala IDE 项目”（http://scala-ide.org/index.html），找到在 Eclipse 中安装和使用 Scala 插件的详细内容。你也可以直接下载一个完整的包含了配置好 Scala 插件的 Eclipse 安装包。

假如你更青睐于使用 Maven 而不是 SBT，那么请访问网页 http://www.scala-lang.org/old/node/345，该页面说明了如何在 Eclipse 中使用 Maven 执行 Scala 构建工作。

使用 Scala 插件与在 Eclipse 中使用 Java 工具非常相似。你可以创建 Scala 项目和文件，执行 SBT 构建和测试，进行代码导航和代码重构。所有这些工作都在 IDE 中完成。

### Eclipse插件

Scala 的 Eclipse 插件提供了一个 Java 插件尚未提供的功能——工作单（worksheet）功能。工作单在提供 REPL 的交互性的同时，还兼具了文本编辑器的便利性。

如果你已经在 Eclipse 中打开了 Scala 项目，那么鼠标右击最上层的项目文件夹，这样便会出现弹出菜单，点击 New -> Other 选项。在 Select a Wizard 对话框中打开 Scala 向导文件夹，选择 Scala Worksheet 选项，并输入工作单的名称和路径。之后，将产生后缀为 .sc 的工作单文件。弹出的工作单中定义了一个默认的对象，而该对象中只包含了一条 `println` 语句。你可以随意对该对象重命名并删除 `println` 语句。

弹出工作单之后，你可以输入你希望执行的语句。每当你保存文件时，文件内容便会被执行一次，而面板右侧则会显示出执行结果。之后你可以再次编辑并存储代码。工作单就像是一个“窗口式”的 REPL 环境。它尤其适合用于对代码片段和 API 进行测试，这其中包括 Java API！

如果你使用的是 IntelliJ IDEA 工具，那么请打开插件首选项（plugins preference）窗口，查找并安装 Scala 插件。该插件提供了与 Eclipse 插件相似的功能，包括 IntelliJ 版本的工作单功能。

最后再提一下，NetBeans 中已经有了一个提供交互式终端功能的 Scala 插件，该插件提供的功能与 Eclipse 和 IntelliJ IDEA 中的工作单功能相似。了解更多 NetBeans 插件的信息，请参考 SourceForg 网站中的相关文章（http://sourceforge.net/projects/erlybird/files/nb-scala/）。

### 文本编辑器

尽管 IDE 在 Scala 开发人员之间很受欢迎，但是我们仍然会发现一些开发人员更青睐于使用文本编辑器，如 Emacs（http://www.gnu.org/software/emacs）、Vim（http://www.vim.org）和 SublimeText（http://www.sublimetext.com）。

你可以查阅你所青睐的编辑器的官方文档及社区论坛，找到一些可用于 Scala 开发的插件和配置选项。许多编辑器插件可以使用 ENSIME 插件（https://github.com/ensime）。ENSIME 插件最初是为 Emacs 编辑器所设计的，它提供了一些“类 IDE”的功能，例如，导航功能和重构功能。

## 21.4 在Scala中应用测试驱动开发

测试驱动开发（test-driven development，TDD）是软件开发中一种已经存在的开发方法，其目的是通过测试驱动代码设计。在 TDD 中，我们需要首先编写包含了一点功能的测试，之后再编写能够使之前编写的测试通过的代码。

不过，TDD 在面向对象工程师当中更受欢迎。函数式编程开发者更倾向于首先使用 REPL 来确定需要选用的类型和算法，再编写代码。如果我们不使用 TDD 的开发流程，不创建一个固定的、可以自动执行“验证”的测试集，这会带来一些不好的地方。不过由于函数式代码的纯粹性，它们很少会发生变化。一些函数式编程开发者会在完成编码工作后再编写一些测试，这样一来也就形成了一套可用于回归测试的测试集。

无论你如何编写操作，你都可以使用 ScalaTest（http://www.scalatest.org）和 Spec2（http://etorreborre.github.io/specs2）这两套库得到支持多种测试风格的 DSL 语言。尤其是

ScalaTest 库，该库通过混入不同的 trait，使你对多种测试风格进行挑选。

对于像 Scala 这样的包含丰富类型系统的函数式语言而言，指定变量类型也可以作为一个测试点，每次执行编译时便会执行相关测试。而测试的目的则是为了尽可能地消除出现无效状态的可能性。

这些类型应该具备定义良好的属性。Haskell 的 Quick-Check（http://www.haskell.org/haskellwiki/Introduction_to_QuickCheck1）测试库使用了基于属性的测试，该测试也被称为基于类型的测试。此类测试也因此流行起来并成为一种独特的测试方式迁移到其他的一些语言中。基于属性的测试会指定类型条件，而该类型的所有实例都必须满足这些条件。我们回顾一下 16.1 节中讨论的内容，基于属性的测试工具会自动生成一些具有代表性的实例，并使用这些实例对条件进行测试。这类测试工具能够验证待测条件是否满足所有的实例（在某些情况下，工具会对这些实例进行组合后再进行测试）。而通常意义的 TDD 工具则不然，TDD 工具会要求测试编写者生成一组具有代表性的实例，并测试所有的可能。

ScalaCheck（http://scalacheck.org）是迁移到 Scala 平台的 QuickCheck 库。ScalaTest 和 Specs2 都可以运行 ScalaCheck 所提供的属性测试。同时，JUnit（http://junit.org）和 TestNG（http://testng.org）库可以使用这两个库，这也使混合 Java 测试和 Scala 测试变得容易起来。

> 假如你日常工作中只使用 Java 语言，而又希望能在不大冒风险的情况下尝试使用 Scala 语言，你可以考虑使用一种或多种 Scala 测试工具，对 Java 代码进行测试驱动。这种行为风险较小，当然也仅限于 Scala 语言。

SBT 现在同时支持这三个工具，而 Ant、Maven 和 Gradle 插件也是如此。

代码示例中出现的测试大都应用了 ScalaTest 和 ScalaCheck 库，在讲解 Java-Scala 互操作时，一些示例使用了 JUnit 工具。

> 这三个工具都可以作为 Scala 内部 DSL 的完美示例。假如希望编写自己的 DSL 语言，你可以学习这些工具，通过学习这些工具的实现我们能学到一些技巧。

## 21.5 第三方库

自第 1 版出版以来，市面上出现了大量的使用 Scala 编写的第三方库。编写本书第一版时某些库并不存在，但现在它们得到了广泛地使用；也有一些库曾经非常流行，现在却已经过时。由于这一趋势还会持续下去，因此本节的内容也只适用于某段时期。而本节谈论的第三方库并不全面。你可以将本节作为学习第三方库的起始点，之后根据需要在 Web 上查找是否有其他可选择的第三方库。

http://typelevel.org 网站中聚集了大量的可用于不同用途的库，你可以从里面找到自己需要的库。

当然，你也可以选择那些使用其他语言编写的 JVM 库。本书将不对这些库进行讲解。

我们将首先关注“全栈”类库和框架，它们用于构造基于 web 的应用。所谓“全栈”，它包括后台服务、HTML 模版引擎、JavaScript 以及 CSS 等所有层面上的类库。而其他的一些第三库使用范围略窄，它们仅局限于某一特定任务。表 21-6 总结了目前最流行的可选方案。

表 21-6：基于web应用的类库

| 库　　名 | URL | 描　　述 |
|---|---|---|
| Play | http://www.playframework.com/ | 由 Typesafe 提供支持的一套全栈框架，该框架提供了 Scala 和 Java API，同时框架集成了 Akka 系统 |
| Lift | http://liftweb.net/ | 第一个 Scala 全栈框架，目前仍然流行 |

表 21-7 列举的这些库都适用于构造后台服务。

表 21-7：服务库

| 库　　名 | URL | 描　　述 |
|---|---|---|
| Akka | http://akka.io | Akka 是一套完备的、基于 actor 的分布式计算系统，我们在 17.3 节中对其进行了讲解 |
| Finagle | https://twitter.github.io/finagle/ | Finagle 是一套用于构建 JVM 服务的可扩展系统，它所构建的 JVM 服务建立在函数式抽象的基础之上。Twitter 开发了这套系统，并将其用于构造一些服务[a] |
| Unfiltered | http://unfiltered.databinder.net/Unfiltered.html | Unfiltered 是一个工具包，它为各类后端 HTTP 请求服务提供了一套统一的 API |
| Dispatch | http://dispatch.databinder.net/Dispatch.html | 提供了同步 HTTP 的相关 API |

a 请参考“函数式系统报告”(http://monkey.org/~marius/sbtb14.pdf)，该网页中包含了最近的一个描述 Finagle 及其设计理念的视频。

表 21-8 描述了各种高级库，这些库利用 Scala 类型系统的某些功能，实现了像函数式编程结构这样的功能。

表 21-8：高级库

| 库　　名 | URL | 描　　述 |
|---|---|---|
| Scalaz | http://scalaz.github.io/scalaz/ | Scalaz 库引导了 Scala 中的范畴理论概念。与此同时，针对一些设计难题，Scalaz 还提供了一些便利的工具。我们在本书的 7.4.4 节和 16.2 节中对该库进行了讲解 |
| Shapeless | https://github.com/milessabin/shapeless | Shapeless 库通过应用类型类和依赖类型，实现了一套范型编程库 |

这两个类都被归为类型级库，而这里描述的其他类型同样应用了一些高级的语言功能。

scala.io 包（http://www.scala-lang.org/api/current/scala/io/package.html）提供的 I/O 功能非常有限，而 Java 相关的 API 则很难操作。表 21-9 列出了两个第三方项目，希望这两个项目能够填补 I/O 库的空白。

表 21-9：I/O 库

| 库　　名 | URL | 描　　述 |
| --- | --- | --- |
| Scala I/O | http://jesseeichar.github.io/scala-io-doc/0.4.3/index.html | 一套具有丰富功能且流行的 I/O 库 |
| Rapture I/O | http://rapture.io/ | 对 `java.io` 接口进行封装，提供了更好的 API |

表 21-10 描述了其他一些库，这些库能够解决某些特定的设计难题。

表 21-10：其他库

| 库　　名 | URL | 描　　述 |
| --- | --- | --- |
| scopt | https://github.com/scopt/scopt | 用于命令行解析的库 |
| Typesafe Config | https://github.com/typesafehub/config | 一套配置库（采用了 Java API） |
| ScalaARM | http://jsuereth.com/scala-arm/ | Joshua Suereth 编写的一套库，用于自动化资源管理 |
| Typesafe Activator | https://github.com/typesafehub/activator | 用于对示例 Scala 项目进行管理，该库位于 http://typesafe.com/activator |

请参考第 18 章的表 18-1，该表列出了一些与大数据和数学相关的库。

最后，我们再回顾一下“前言”中的内容。在前言中，我们提到 Scala 2.11 版对类库进行了模式化处理，并将类库解耦成一个个较小的 JAR 文件，从而使得较少使用的类组件被设置成可选组件。表 21-11 对可选组件进行了描述，你也可以在 Maven 资源库（http://mvnrepository.com/artifact/org.scala-lang.modules）中找到这些组件。

表 21-11：Scala 2.11可选模块

| 库　　名 | 组件(Artifact)名 | 描　　述 |
| --- | --- | --- |
| XML | scala-xml | 用于构造及解析 XML |
| Parser Combinators | scala-parser-combinators | 用于构造解析器的组合库 |
| Swing | scala-swing | Swing 库 |
| Async | scala-async | Scala 同步编程的辅助库，提供了能够直接操作 Future 类型的 API |
| Partest | scala-partest | 一套适用于编译器和类库的测试框架 |
| Partest Interface | scala-partest-interface | 一套适用于编译器和类库的测试框架 |

了解目前最全面的第三方库列表，请访问“Github 里最棒的 Scala 库列表”（https://github.com/lauris/awesome-scala）。除了这个列表之外，http://ls.implicit.ly/ 同样集合了一批 Scala 库。

## 21.6　本章回顾与下一章提要

我们在本章对每天都会使用的 Scala 工具进行了详细地讲解。下一章中，我们将学习 Java 和 Scala 代码是如何进行互操作的。

# 第22章

# 与Java的互操作

在所有的 JVM 语言中，Scala 与 Java 代码的互操作性是其中最完美的。本章首先讨论 Scala 代码与 Java 代码的互操作性问题。

因为 Scala 语法基本上是 Java 语法的一个超集，从 Scala 中调用 Java 代码通常很简单。反过来调用，你需要了解一些 Scala 特性是如何编码为字节码并满足 JVM 规范的。本章将讨论一些互操作性问题。

## 22.1 在Scala代码中使用Java名称

Java 对类型、方法、字段和变量名的规定比 Scala 更加严格。所以，几乎所有情况下，你都可以在 Scala 代码中使用 Java 名称，创建 Java 类型的新实例，调用 Java 方法，使用 Java 变量与实例字段。

唯一的例外是，Java 的名字实际上是 Scala 的关键字的情况。正如我们在 2.7 节看到的情况，我们需要使用反引号进行“转义”。例如：Scala 中的 `match` 关键字和 `java.util.Scanner`（http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html）中的 `match` 方法同名，调用该方法时要用 `myScanner.`match`` 的形式。

## 22.2 Java泛型与Scala泛型

一直以来，我们都可以在 Scala 代码中使用 Java 类型，如 `java.lang.String`。你甚至可以使用 Java 的泛型类型，如：在 Scala 中使用 Java 的集合。

那么如何在 Java 中使用 Scala 的参数化类型呢？考虑下面的 JUnit 4 测试代码，该代码使用了 `scala.collection.mutable.LinkedHashMap`（http://www.scala-lang.org/api/current/index.

html#scala.collection.mutable.LinkedHashMap） 和 `scala.Option`（http://www.scala-lang.org/api/current/index.html#scala.Option）。这里显示了一些你可能会遇到的特性：

```
// src/test/java/progscala2/javainterop/SMapTest.java
import org.junit.*;
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;
import static org.junit.Assert.*;
import scala.*;
import scala.collection.mutable.LinkedHashMap;

public class SMapTest extends org.scalatest.junit.JUnitSuite {     // ❶
  static class Name {
    public String firstName;
    public String lastName;

    public Name(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName  = lastName;
    }
  }

  LinkedHashMap<Integer, Name> map;

  @Before
  public void setup() {
    map = new LinkedHashMap<Integer, Name>();
    map.update(1, new Name("Dean", "Wampler"));
  }

  @Test
  public void usingMapGetWithOptionName() {                        // ❷
    assertEquals(1, map.size());
    Option<Name> n1 = map.get(1);  // Note: Option<Name>
    assertTrue(n1.isDefined());
    assertEquals("Dean", n1.get().firstName);
  }

  @Test
  public void usingMapGetWithOptionExistential() {                 // ❸
    assertEquals(1, map.size());
    Option<?> n1 = map.get(1);    // Note: Option<?>
    assertTrue(n1.isDefined());
    assertEquals("Dean", ((Name) n1.get()).firstName);
 }
}
```

❶ 如果混入了 `JUnitSuite`，以上的 JUnit 测试代码将由 ScalaTest 运行。

❷ 显式地使用类型 Name。

❸ 获取 n1 后将其转为目前已存在的类型 Name。

你也可以使用 Scala 的元组类型，不过不能使用 Scala 的语法糖，如 `("someString", 101)`：

```
// src/test/java/progscala2/javainterop/ScalaTuples.java
package progscala2.javainterop;
import scala.Tuple2;

public class ScalaTuples {
  public static void main(String[] args) {
    Tuple2 stringInteger = new Tuple2<String,Integer>("one", 2);

    System.out.println(stringInteger);
  }
}
```

然而，在 Java 中使用 FunctionN 类型是非常困难的，因为编译器会自动合成“隐藏”的成员。例如，试图编译下面的代码将会失败：

```
// src/test/java/progscala2/javainterop/ScalaFunctions.javaX
package progscala2.javainterop;
import scala.Function1;

public class ScalaFunctions {
  public static void main(String[] args) {
    // Fails to compile, due to missing methods the Scala compiler would add.
    Function1 stringToInteger = new Function1<String,Integer>() {
      public Integer apply(String s) {
        Integer.parseInt(s);
      }
    };

    System.out.println(stringToInteger("101"));
  }
}
```

编译器会报错，提示抽象方法 apply$mcVJ$sp(long) 未定义。Scala 编译器会为我们自动生成，但 Java 编译器则不会。

这严重限制了在 Java 中对 Scala 集合的高阶函数的调用。你可能想尝试通过 Java 8 的 Lambda 来代替 scala.FunctionN，但它们是不兼容的。（Scala 2.12 计划将 Scala 的 Function 和 Java 的 Lambda 表达式进行统一，以消除这种不兼容问题。）

因此，如果想从 Java 调用 Scala 的 API，那就不能调用高阶方法，高阶方法即这些方法的参数或返回值是一个函数。

## 22.3 JavaBean的性质

我们在第 8 章中看到，Scala 为了支持统一访问原则，并没有遵循 JavaBean 对字段读写方法的约定。

但是，有时你的确需要 JavaBean 风格的访问方法。例如：一些依赖注入框架需要用到它们。另外，支持“自省”的 IDE 也需要使用这些方法。

Scala 通过一个可以应用在字段上的标记 @scala.beans.BeanProperty（http://www.scala-lang.org/api/current/scala/beans/BeanProperty.html）解决了这个问题，该标记告诉编译器生

成 JavaBean 风格的 getter 和 setter 方法。此外，`scala.beans` 包（http://www.scala-lang.org/api/current/scala/beans/package.html）还包含其他用于配置 bean 属性的标记。

在 Scala 2.10 及之前的版本中，这个用来添加 JavaBean 标记的包实际上叫 `scala.reflect`。

例如，我们可以对之前见过的 `Complex` 类的字段做标记：

```
// src/main/scala/progscala2/javainterop/ComplexBean.scala
package progscala2.javainterop

// Scala 2.11. For Scala 2.10 and earlier, use scala.reflect.BeanProperty.
case class ComplexBean(
  @scala.beans.BeanProperty real: Double,
  @scala.beans.BeanProperty imaginary: Double) {

  def +(that: ComplexBean) =
    new ComplexBean(real + that.real, imaginary + that.imaginary)
  def -(that: ComplexBean) =
    new ComplexBean(real - that.real, imaginary - that.imaginary)
}
```

这个类已经被 SBT 编译了。如果你反编译 ComplexBean.class 文件，可以在输出中找到以下方法：

```
$ javap -cp target/scala-2.11/classes javainterop.ComplexBean
...
  public double real();
  public double imaginary();
  ...
  public double getReal();
  public double getImaginary();
  ...
}
```

因为这些字段是不可变的，这里没有 setter 方法。与此相反，对原版 `Complex` 做反编译，只会得到 `real()` 方法和 `imaginary()` 方法。即使使用了 `BeanProperty` 标记，你得到的仍然是普通的读方法和可能的写方法。

## 22.4 AnyVal类型与Java原生类型

注意在之前的 `Complex` 示例中，`Double` 类型的字段都被编译成 Java 原生的 `double`。所有 `AnyVal` 类型被转换为它们相应的 Java 原生类型。特别的是，`Unit` 被映射为 `void` 类型。

## 22.5 Java代码中的Scala名称

Scala 对标识符的限定更灵活，例如：`*`、`<` 等 Scala 操作符在字节码中不允许用来做标识符。因此，这些字符经过编码（或称为“变形”，mangled），以满足 JVM 的限制。其对应关系如表 22-1 所示。

表22-1：操作符的编码规则

| 操作符 | 编码结果 | 操作符 | 编码结果 | 操作符 | 编码结果 | 操作符 | 编码结果 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| = | $eq | > | $greater | < | $less | | |
| + | $plus | - | $minus | * | times | / | $div |
| \ | $bslash | \| | $bar | ! | $bang | ? | $qmark |
| : | $colon | % | $percent | ^ | $up | & | $amp |

## 22.6 本章回顾与下一章提要

Scala 的一个重要优点是，你可以继续使用现有的 Java 代码。从 Scala 调用 Java 非常容易（只有少数例外）。

为了真正成功地使用 Scala 完成应用程序，下一章将涵盖设计中应该考虑的注意事项。

# 第23章

# 应用程序设计

到目前为止，我们已经讨论了大多数的语言特性。这些特性让我们编写的应用程序变得非常简短，即使是编写第 18 章中的应用也是如此。这是一件很棒的事情。代码行数的显著减少意味着软件开发时碰到问题的数量也会明显减少。

不过，并不是所有的应用程序都会变得那么精简。本章将思考如何构造大型应用程序。我们会讨论到一些之前未提及的语言和 API 特性，除此之外，还会涉及一些设计模型和习语，以及一些架构方法。这些有助于我们理解如何将 trait 作为模块来使用，以及如何在面向对象设计和函数式设计技巧中达到平衡。

## 23.1 回顾之前的内容

下面我们将复习之前已经学习到的一些概念。当遇到一些小的设计难题时，通过应用学到的概念，我们能够更容易地解决这些问题。因此，这些概念提供了解决应用程序问题的一些稳定的基础方法。

- *函数式容器*

  本书的大多数示例都比较简短，这归功于集合和其他容器所提供的简洁且强大的组合器。这些组合器让我们仅仅通过一些极为简短的代码就可以实现逻辑功能。

- *类型*

  类型会引入约束。理想情况下，类型能提供与程序行为相关的尽可能多的信息。举个例子，使用 `Option`（http://www.scala-lang.org/api/current/index.html#scala.Option）会消除对 `null` 的使用。稍后，我们在列举的错误处理策略中也会提到相关的内容。我们还可以利用参数化类型成员和抽象类型成员进行抽象和代码重用，例如：我们在 2.13 节中

曾经列举了 `Reader` 抽象体的示例，在该示例中引入了类族多态性（也被称作协变特化，convariant specialization）。

- 混入 trait

  trait 能够将行为模式化，也可以将多个行为组合起来（请参考第 3 章 3.14 节的内容和第 9 章的相关内容）。

- `for` 推导式

  `for` 推导式与容器结合起来，并辅以 `flatMap`、`map` 和 `filter/withFilter` 方法，形成了一门便利的 DSL 语言（详见第 7 章）。

- 模式匹配

  模式匹配能够快速地提取数据，以进行后续的数据处理（详见第 4 章）。

- 隐式

  隐式能够解决一些设计难题，包括减少代码量、通过方法调用切换线程上下文、隐式转换以及一些类型约束（详见第 5 章）。

- 细粒度可见性规则

  Scala 提供了细粒度可见性规则，能够对 API 实现细节的可见性进行精准地控制，只有客户需要使用的公有抽象才会对用户开放。尽管在设定可见性规则时需要遵循某些准则，不过运用这些准则能够避免 API 内部出现可预防耦合，而这些耦合会使得架构演变变得更加复杂，因此花费这些精力是值得的（请参考第 13 章的内容）。

- 包对象

  包对象法不同于细粒度可见性控制，它将所有的实现结构都放置到一个受保护的包内，之后再提供一个顶级包对象，该对象只对外暴露适当的公用抽象体。例如：对于那些应该被隐藏的类型而言，我们可以使用类型成员作为其别名，将其隐藏（请参考 2.12.2 节的相关内容）。

- 错误处理策略

  `Option`（http://www.scala-lang.org/api/current/index.html#scala.Option）、`Either`（http://www.scala-lang.org/api/current/scala/util/Either.html）、`Try`（http://www.scala-lang.org/api/current/scala/util/Try.html）以及 Scalaz 提供的 `Validation`（http://docs.typelevel.org/api/scalaz/stable/7.0.4/doc/#scalaz.Validation）类型都将异常和其他错误具化为特定类型，使得这些异常错误都能作为“正常”结果值从方法中返回。而类型签名则能告知用户该方法所返回的成功或失败的结果（请参考 7.4 节的内容）。

- `Future`

  为了能够对错误进行处理，`Future` 类（http://www.scala-lang.org/api/current/scala/concurrent/Future.html）很好地利用了 `Try` 类型。Akka（http://akka.io）实现的 actor 模型提供了一个强壮的、具有策略性的模型，该模型可用于对 actor 进行监管并处理失败的场景（参见第 17 章的内容）。

接下来，我们将思考其他应用级的问题，首先是注解。

## 23.2 注解

注解（annotation）是为声明体添加元数据的一项技术，这门技术存在多个语言中。一些 Scala 注解还为编译器提供了指令。这些注解与对象关系映射（ORM）框架联合使用，为类型制定持久化的映射信息。注解还可用于依赖注入（dependency injection，DI），举个例子，我们可以使用注解将多个组件编织到一起（Martin Fowler 在之前的博文 http://bit.ly/1zmlaf1 中，曾经举过相关的示例）。我们之前也使用了一些注解，如 `scala.annotation.tailrec`（http://www.scala-lang.org/api/current/index.html#scala.annotation.tailrec）。假如认为某个递归函数实现了尾递归，我们可以为该函数添加 `tailrec` 注解；如果该函数并未实现尾递归，该注解可以发现这一错误。

在 Scala 中，注解的使用并不如 Java 语言中注解出现得那么频繁，不过它们还是有用的。Scala 使用注解实现了一些 Java 关键字（如 `strictfp`、`native` 关键字）。Scala 代码还能使用 Java 注解。如果愿意，你可以使用 Spring 框架（http://spring.io）或 Guice 框架（https://github.com/google/guice）中的注解实现依赖注入。

Scala 的注解都继承自 `scala.annotation.Annotation`（http://www.scala-lang.org/api/current/#scala.annotation.Annotation）类型。那些直接继承自该抽象类的注解无法被系统保留，因此类型检查器无法使用这些注解，而运行时也无法调用它们。但有两个主要的 Annotation 子类型（trait）突破了这些限制。继承自 `scala.annotation.ClassfileAnnotation`（http://www.scala-lang.org/api/current/index.html#scala.annotation.ClassfileAnnotation）的注解将作为 Java 注解存储在类文件中，而继承自 `scala.annotation.StaticAnnotation`（http://www.scala-lang.org/api/current/index.html#scala.annotation.StaticAnnotation）的注解则允许类型检查器对其进行访问，即使是需要跨编译单元，它也允许类型检查器对其访问。

表 23-1 列举了直接继承自 `Annotation` 类型的注解（包括 `ClassfileAnnotation` 和 `StaticAnnotation`）。

**表23-1：继承了`Annotation`类型的Scala注解**

| 注解名 | Java中对应的注解 | 描　　述 |
| --- | --- | --- |
| `ClassfileAnnotation` | Annotate with`@Retention (RetentionPolicy.RUNTIME)` | 所有需要以 Java 注解的形式存储在类文件中，以供运行时访问的注解都应该继承 ClassfileAnnotation 这一 trait |
| `BeanDescription` | `BeanDescriptor` (class) | 该注解可用于 JavaBean 类型或成员，它会为修饰的类型或成员附加上一条简短的描述信息（以注解参数的方式提供），当生成 bean 信息时，这条简短的描述信息将会被包含在 bean 信息中 |
| `BeanDisplayName` | `BeanDescriptor` (class) | 该注解可用于 JavaBean 类型或成员，它会为修饰的类型或成员附加上名称（以注解参数的方式提供），当生成 bean 信息时，该名称将会被包含在 bean 信息中 |

（续）

| 注解名 | Java中对应的注解 | 描　　述 |
|---|---|---|
| BeanInfo | BeanInfo (class) | 该注解起到了标记的作用，凡是使用了这一注解的 Scala 类都应该为其生成一个 BeanInfo 类。在 BeanInfo 类中，val 变量被生成为只读属性，var 变量被生成为读写属性，而 def 变量则被生成为函数 |
| BeanInfoSkip | *N.A.* | 该注解起到了标记的作用。使用该注解的成员不应生成对应的 bean 信息 |
| StaticAnnotation | 静态字段 @Target(ElementType.TYPE) | 对于任何注解而言，如果该注解希望能够跨编译单元可见并定义“静态”的元数据，那么它需要将 StaticAnnotation 特征作为其父特征 |
| TypeConstraint | *N.A.* | TypedConstraint 是一个注解 trait，它可以作用于一些为类型定义约束的 trait 之上。无需使用定义或使用类型时的外部信息，仅依靠类型自身所提供的信息，TypedConstraint 便能完成其工作。编译器会利用这一限制对约束进行重写 |
| unchecked | 类似于 @SuppressWarnings("unchecked") | unchecked 是一个标记注解，我们可以将其用于 match 语句的选择器（例如：x match {...} 中的 x）中。假如 match 语句中的 case 表达式不能覆盖“所有可能”，unchecked 注解会阻止编译器抛出警告 |

表 23-2 列出了 StaticAnnotation 的子类型。除了定义在 scala.annotation.meta 包中的子类型之外，所有类型都包含在下列列表中。而 scala.annotation.meta 包中的子类型则单独列在表 23-3 中。

**表23-2：继承自StaticAnnotation的Scala注解[a]**

| 注解名 | Java中对应的注解 | 描　　述 |
|---|---|---|
| BeanProperty | JavaBean convention | 该注解可以用作字段标记（构造器参数中使用 val 关键字和 var 关键字），编译器因此生成 JavaBean 格式的 getter 和 setter 方法。编译器只会对使用 var 声明的变量生成 setter 方法。请参考 22.3 节的相关讨论 |
| BooleanBeanProperty | *same* | 与 BeanProperty 类似，不同的地方在于 BooleanBeanProperty 的 getter 方法名是 isX，而不是 getX |
| cloneable | java.lang.Cloneable(interface) | 起到类标志的作用，表明该类可以被克隆 |

a：此处并未列举 annotation.meta 包内定义的注解，我们会在下面的表中列举这些注解。

（续）

| 注解名 | Java中对应的注解 | 描　述 |
|---|---|---|
| compileTimeOnly | *N.A.* | 编译结束后，这类注解项便不再可见。例如：这类注解项可用在宏中，当对宏执行完扩展操作后，这些注解项便消失了 |
| deprecated | java.lang.Deprecated | deprecated 注解可以用在任何类型的定义体上。它表明该“定义”项已经过时。使用该定义项时，编译器会抛出警告信息 |
| deprecatedName | *N.A.* | deprecatedName 注解标注了参数名已经过时。由于调用代码会使用过时的参数名，因此该注解是需要的。例如 val x = foo(y = 1) |
| elidable | *N.A.* | 用于阻止代码生成。例如，该注解可以用于阻止产生不需要的日志消息 |
| implicitNotFound | *N.A.* | 定制无法找到隐式值时的错误消息 |
| inline | *N.A.* | 用作方法标志，当看到某个方法使用了 inline 注解时，编译器应该“尽力”将该方法内联（inline） |
| native | native ( 关键字 ) | native 注解对方法进行标注，表明该方法会使用“本地”方法实现。编译器不会负责生成该方法的方法体，不过使用该方法时会执行类型检查 |
| noinline | *N.A.* | noinline 是一个方法标志，它能够阻止编译器对被标注方法进行内联（inline），即使对该方法执行内联操作看上去是安全的 |
| remote | java.rmi.Remote(interface) | 该注解用作类标志，表明该类应该允许远程 JVM 调用 |
| specialized | *N.A.* | 作用于参数化类型和参数化方法中的类型参数。该注解要求编译器根据平台原始类型生成其对应的 AnyVal 方法或类型时，对生成的代码进行优化。你还能选择是否对生成的特化实现的 AnyVal 类型进行限定 |
| strictfp | strictfp (keyword) | 严格执行浮点语义 |
| switch | *N.A.* | switch 注解作用于 match 表达式中，例如 (x:@switch) match {...}。当出现 switch 注解时，编译器会检查该 match 语句是否已经被编译成某个基于表或可查找的 switch 语句。假如验证失败，意味着该 switch 语句被编译成了一组条件判断语句。由于该组条件语句效率低下，编译器会抛出错误 |
| tailrec | *N.A.* | tailrec 是一个方法类注解，它要求编译器对该方法进行验证，确认编译该方法时使用了尾调用优化（tail-call optimization）。当出现 tailrec 注解时，假如该方法无法被优化到一个循环内，那么编译器将抛出错误。假如使用了 tailrec 注解的方法是可重写的，由于该方法不是 private 或 final 方法，编译器也会抛出错误 |

（续）

| 注解名 | Java中对应的注解 | 描　　述 |
| --- | --- | --- |
| `throws` | `throws` (keyword) | 该注解用于对方法进行标注，它描述了被标注的方法可能会抛出的异常。详细信息请参考后续文章的相关内容 |
| `transient` | `transient` (keyword) | 用于对字段进行标注，表示该字段是瞬态的（即非持久化的） |
| `unchecked` | *N.A.* | 限制编译器执行检查，如检查 match 表达式是否完整 |
| `uncheckedStable` | *N.A.* | `uncheckedStable` 用于对值进行标记。被标记的值即使是可变类型，也会被视为是稳定的 |
| `uncheckedVariance` | *N.A.* | 可用于可变类型参数的注解。你也可以对类型化参数使用该注解，以关闭型变检查 |
| `unspecialized` | *N.A.* | 对特化类型生成进行限制 |
| `varargs` | *N.A.* | 如果被修饰的方法中包含了一些重复的参数，那么考虑互操作性，编译器会为其生成 Java 风格的 `varargs` 方法 |
| `volatile` | `volatile` (keyword, for fields only) | 可用于某一单独字段，表明可能会有某个单独的线程对该字段进行修改。你也可以对整个类型施加该注释，这样一来，该类型的所有字段都会受到影响 |

`annotation.meta` 包（http://www.scala-lang.org/api/current/index.html#scala.annotation.meta.package）中定义了一些额外的 `StaticAnnotation` 类型，这些注解能够以较小的粒度对字节码中的注解应用进行控制。

表 23-3：Scala meta注解

| 注解名 | 描　　述 |
| --- | --- |
| `beanGetter` | 对 `@BeanProperty` 注解进行限制，使其只出现在生成的 getter 方法中（例如字段 `x` 生成的 `getX` 方法） |
| `beanSetter` | 对 `@BeanProperty` 注解进行限制，使其只出现在生成的 setter 方法中 |
| `companionClass` | Scala 编译器会为对应的隐式类生成隐式转换方法 |
| `companionMethod` | 与 `companionClass` 注解相似，不过 `companionMethod` 会同时将该注解功能应用到生成的转换方法上 |
| `companionObject` | 创建 `companionObject` 的本意是希望将其用到自动生成的伴生对象的 case 类中。现在看来，该注解没有什么作用 |
| `field` | `field` 可用于那些指定默认目标的注解定义中，这里的默认目标专指字段。我们可以使用该表之前提及的注解对默认目标进行覆写 |
| `getter` | 与 `filed` 相似，不过只提供了 getter 方法 |
| `languageFeature` | 用于提供 `scala.language` 包中定义的语言功能 |
| `param` | 与 `field` 相似，不过只提供了 param 方法 |
| `setter` | 与 `field` 相似，不过只提供了 setter 方法 |

最后，表 23-4 列举了继承自 ClassfileAnnotation 注解的唯一子类型。

表23-4：继承自ClassfileAnnotation的Scala注解

| 注解名 | Java中对应的注解 | 描述 |
|---|---|---|
| SerialVersionUID | serialVersionUID static field in a class | 为了能够将对象序列化，该注解定义了一个全局唯一的 ID。该注解的构造器接受用户输入一个 Long 类型的参数，而该参数将用于生成 UID |

在 Scala 中定义注解并不需要使用 Java 类似的特殊语法，下面列出 implicatNotFound（http://www.scala-lang.org/api/current/index.html#scala.annotation.implicitNotFound）的实现代码：

```
package scala.annotation

final class implicitNotFound(msg: String) extends StaticAnnotation {}
```

## 23.3 Trait即模块

Java 将类和包都作为模块化的具体单元，而 JAR 包是最常用的粗粒度组件抽象。一直以来，对可见性的控制能力有限是包的一大短板。包作为模块化单元无法对外隐藏 public 可见性的实现类型，因此很少有用户利用包进行模块化。Scala 提供了丰富的可见性规则，这也使得包在 Scala 中能够作为模块化单元，不过这种做法并未得到广泛应用。其实，包对象提供了另一种区分客户可执行操作和不允许执行操作的方法。

提供组合功能是模块化的另一个重要目标。如我们所见，Scala 的 trait 能够完美地支持组件混入。事实上，相较于类，Scala 更青睐于将 trait 用作定义模块的机制。

我们在 14.6 节中使用 Cake 模式草拟了一个示例。下面列出了该示例的重要部分：

```
// src/main/scala/progscala2/typesystem/selftype/selftype-cake-pattern.sc

trait Persistence { def startPersistence(): Unit }              // ❶
trait Midtier { def startMidtier(): Unit }
trait UI { def startUI(): Unit }

trait Database extends Persistence {                             // ❷
  def startPersistence(): Unit = println("Starting Database")
}
trait BizLogic extends Midtier {
  def startMidtier(): Unit = println("Starting BizLogic")
}
trait WebUI extends UI {
  def startUI(): Unit = println("Starting WebUI")
}

trait App { self: Persistence with Midtier with UI =>           // ❸
  def run() = {
    startPersistence()
    startMidtier()
    startUI()
```

```
  }
}

object MyApp extends App with Database with BizLogic with WebUI    // ❹
```

❶ 定义了三个 trait，分别对应了应用程序的持久化层、中间层和 UI 层。

❷ 实现了各个 trait 的“具体”行为。

❸ 定义了一个 trait（也可以使用抽象类来表示），该 trait 中定义了将各层代码粘合在一起的“骨架”代码。为了简单起见，`run` 方法只简单地启动各个职责层。

❹ 定义了 `MyApp` 对象，`MyApp` 对象继承了 `App` 对象并混入了三个具体 trait，这三个 trait 均实现了所需的行为。

`Persistence`、`Midtier` 和 `UI` 特征均起到了模块抽象体的作用。而具体的实现则很清楚的与这些定义分隔开了。将这些模块抽象体与具体实现组合起来，便构成了这个应用程序。自类型注解指定了组合的规则。

在这个示例中，作为依赖注入机制（http://jonasboner.com/2008/10/06/real-world-scala-dependency-injection-di）的替代品，Cake 模式很好地发挥了其作用。在构造 Scala 自身的编译器时也使用了 Cake 模式（参见 Martin Odersky 和 Matthias Zenger 在 OOPSLA’05 会议上发表的文章 *Scalable Component Abstractions*）。

不过，Cake 模式也有一些弊端。假如 Cake 模式中“蛋糕”之间的依赖关系太过复杂，可能会导致各个依赖之间的初始化顺序出现问题。解决方法包括使用 lazy val、和使用方法取代字段，这两个解决方法都能对初始化事件进行延迟，直到依赖于该“蛋糕”的其他“蛋糕”被初始化才会触发该事件。

在一些应用程序中，Cake 模式的作用已经不再那么明显，甚至在编译器中也是如此。不过 Cake 模式仍有用武之地，只是在使用时需要谨慎。

## 23.4 设计模式

批评者虽然认可设计模式能够实现语言自身所缺失的一些功能，但是设计模式最近还是遭受了一些抨击。“四人组”所提出的一些模式[1]实际上在 Scala 中并不需要，这是因为 Scala 语言自身所提供的功能已经提供了更好的解决方法。而其他的一些模式则是作为语言的一部分而存在，我们无需编写特别的代码来实现这些模式。当然，设计模式常被误用和滥用，以致于变成每个设计难题的潘多拉之盒，不过这并不是设计模式的过错。

设计模式将那些经常出现、广泛使用的有用思想文档化。模式也变成了一个有用的词汇，开发者们可以使用模式进行交流。在 16.2 节中，我提到过，分类其实是一种设计模式。我们从数学中汲取了该模式的思想，并将其运用到函数式编程中。

下面我们将列举“四人组”所总结出的那些模式，并讨论 Scala 和 Akka 这样的工具集中对应的实现，我们会将这些实现作为设计模式的应用示例（无论 Scala 中是否出现过这些设

注 1：请参考 Erich Gamma 等人编写的《设计模式：可复用面向对象软件的基础》。

计模式的名字)。本书在讲解设计模式时，将遵循下列分类：创建型模式、结构型模式和行为型模式。

## 23.4.1 构造型模式

- 抽象工厂（abstract factory）

  抽象工厂是一种基于特定类型家族构造实例的抽象体，使用抽象工厂时无需指定构造实例的类型。对象中的 `apply` 方法能实现这一模式，`apply` 方法能够根据传入的参数初始化出适当类型的实例。除此之外，将函数传递给 `Monad.flatMap`，或者使用 Applicative 定义 `apply` 方法同样能够实现这种抽象构造的功能。

- 构造器模式（builder）

  构造器会根据对象内容，对复杂对象的构造过程进行分解，这样一来便可以在构造多种不同的对象时复用相同的构造过程。`collection.generic.CanBuildFrom`（http://www.scala-lang.org/api/current/index.html#scala.collection.generic.CanBuildFrom）便是一个很好的 Scala 示例，使用该对象时可以利用像 `map` 这样的组合器方法构造一个相同类型的新的集合对象，构造出的集合类型与输入集合类型相同。

- 工厂方法（factory method）

  在父类中定义一个方法，子类型会重载（或实现）该方法。而子类型正是通过这种方式决定初始化什么类型和初始化的方式。`CanBuildFrom.apply` 方法便是一个用于创建构造器实例的抽象方法，而新创建的构造器则可以用于构建实例。子类型与特定的实例可以提供工作方法的具体实现。`Applicative.apply` 可以提供类似的抽象。

- 原型模式（prototype）

  在原型模式中存在一个用作原型的实例，之后复制该原型实例并对其进行选择性的修改，从而得到新实例。Case 类的 `copy` 方法是原型模式的一个很好的示例，使用 `copy` 方法时，用户可以对某一实例进行克隆，同时用户还能通过指定参数的方式对该实例进行修改。我们在 16.2 节中提到了 Lens（函数式访问器），不过并未对 Lens 进行讲解。与其他属性访问器相比，Lens 提供了另一种设置（并复制）或获取任意深度对象图谱中对象值的技术。

- 单例模式（singleton）

  单例模式确保类型只有一个实例，且该类型的所有用户都能访问这个唯一的实例。Scala 通过 `object` 实现了单例模式，因此该模式可以看作 Scala 语言的最好功能。

## 23.4.2 结构型模式

- 适配器模式（adapter）

  适配器模式创建用户期望的接口对其他抽象体进行封装，封装完成后，用户便可以使用该抽象体。在之前的 9.2 节和 14.7 节中，我们讨论了 Observer 模式的一些可能的实现，以及如何在这些实现中进行取舍，其中重点提到了抽象体和潜在观察者之间建立耦合的方式。我们首先使用 trait 描述了观察者需要实现的功能。之后我们为了降低依赖，又使用结构化类型取代了该 trait。事实上，潜在观察者并不需要实现我们所定义的 trait，

它只需要提供某一专门的方法即可。最后，我们注意到如果使用匿名函数，可以完全地解除与观察者之间的耦合。这个匿名函数便是适配器。观察者所观察的主体会调用该适配器，而该适配器内部则会调用所有需要执行的观察者。

- 桥接模式（bridge）

  将抽象体和实现体分隔开，这样一来便能独立地对这两者进行更改。类型类（type class）可以看作是桥接模式的一个有趣示例，从逻辑上看，桥接模式被类型类应用到了极致。类型类不仅将类型可能需要的抽象从类型中剥离开，只在需要时才重新添加到类型中，而且指定类型的类型类抽象实现同样也被单独定义在其他地方。

- 组合模式（composite）

  使用由一组实例组成的树形结构表示“局部－整体”的层次关系。对个体实例和组合实例进行相同的对待。函数式代码倾向于避免使用类型之间的各类层次关系，它青睐于使用像树这样的泛型结构来表示层次，并提供操作树的统一访问方法和完整的组合器。Lens 便是这样一类工具，我们可以使用它对复杂的组合进行处理。

- 装饰模式（decorator）

  为某一对象“动态”地附加额外的职责。无需对类型的原始代码进行修改，类型类在编译期间便能执行这样的操作。假如你希望能够提供真正的运行灵活性，那么 `Dynamic` 特征（http://www.scala-lang.org/api/current/index.html#scala.Dynamic）便能派上用场。假如你希望能够对某一值或计算过程进行“装饰”，那么你可以分别使用 Monad 和 Applicative。

- 外观模式（facade）

  为了能够使子系统更易使用，外观模式为子系统中的多个接口提供统一的接口。包对象便支持这类模式。包对象能够只对外暴露应该公有的类型和行为。

- 享元模式（flyweight）

  为了能够有效地允许大量细粒度对象对资源进行访问，采用共享的方式提供资源。函数式编程强调对象的不可变性，这也使得我们能够很直接的在函数式编程语言中实现享元模式。像 `Vector`（http://www.scala-lang.org/api/current/index.html#scala.collection.immutable.Vector）这样的持久化数据类型便是很重要的示例。

- 代理模式（proxy）

  代理模式会提供其他实例代理对象，以对该实例的所有访问进行控制。包对象在细粒度级别上提供了对代理模式的支持。值得注意的是，由于多个用户操作不可变实例时并不会出现访问冲突这样的问题，因此 Scala 对访问控制的需求也就降低了。

### 23.4.3 行为型模式

- 职责链模式（chain of responsibility）

  职责链模式消除了发送者和接收者之间的耦合。该模式允许一组潜在的接收者对请求进行处理。当前面的接收者执行完毕，后续的接收者才开始执行。这也是模式匹配的工作原理。职责链模式的描述更适用于 Akka 的 `receive` 块，在 `receive` 代码块中，“发送者”和“接收者”不只是简单的暗喻。

- 命令模式（command）

  对服务请求进行具体化。这样一来，我们便能将这些请求队列化，使请求可以执行撤销、重新执行等操作。这也是 Akka 的工作方式。尽管尚未支持撤销和重做功能，不过 Akka 在原则上是能够提供这些功能的。Monad 模式的一类典型应用便是此类问题的一种扩展，在此类应用中，我们会仔细管理系统的状态转换，并将“命令”步骤按照可预测的顺序进行组合（对于默认情况下惰性求值的语言而言，这是一项重要的特性）。

- 解释器模式（interpreter）

  定义了一门语言以及解释该语言表达式的方式。“四人组”编写了设计模式后，DSL 这一名词也开始兴起。DSL 语言中的一些方法已经在第 20 章讲解过。

- 迭代器模式（iterator）

  能够在不暴露容器实现细节的情况下对该容器进行遍历。操作函数化容器时，几乎所有的操作都遵循该设计模式。

- 中介者模式（mediator）

  为了避免实例之间直接地交互，中介者模式使用中介者来实现交互功能，该模式允许各类交互分别独立地改良。`ExecutionContext`（http://www.scala-lang.org/api/current/index.html#scala.concurrent.ExecutionContext）被用于协调异步计算。在使用 `Future` 时，正是由于 `ExecutionContext` 的存在，`Future` 对象无需了解关于异步协作的任何机制。因此我们可以将 `ExecutionContext` 视为中介者的一个例子。与之类似，Akka 运行时通过在 actor 之间建立少量的连接，能对 actor 之间的消息进行调解。在此期间，Akka 需要使用特定的 `ActorRef` 对象（http://doc.akka.io/api/akka/current/index.html#akka.actor.ActorRef）发送消息，无需通过程序硬编码描述 actor 之间的依赖关系，Akka 只要利用像名字查找这样的方法便能找到消息的接收方。除此之外，Akka 为 actor 提供了一层可以进行间接交互的模块。

- 备忘录模式（momento）

  备忘录模式会捕获实例状态，对实例状态进行存储，并使用存储状态对该状态进行恢复。使用纯函数能够更容易地实现记忆化（memoization）功能。我们可以使用装饰器（decorator）添加备忘录，这样一来，假如添加备忘录时传入了之前已经使用过的参数，系统可以避免重复调用该函数，直接返回备忘录。

- 观察者模式（observer）

  根据主体状态，建立主体与观察者之间一对多的依赖关系。当主体状态发生变化时，通知观察者。在之前讲述适配器模式时，我们曾讨论过该模式。

- 状态模式（state）

  当实例状态发生改变时，状态模式允许实例对自身行为进行更改。假如状态值是不可变值，那么为了能够表示新的状态，状态模式会构造新的实例。原则上，新构造的实例会表现出不一样的行为，尽管这些变化会受到某个常见父类型抽象体的限制。状态机是状态模式的更常见的形式。我们在 17.3 节中曾经看到过状态机，其中 Akka 的 actor 和 actor 模型能够实现通常意义上的状态机。

- *策略模式*（strategy）

  对一组相关算法进行物化操作，使得这些算法能交换使用。利用高阶函数能够简单地实现这一模式。例如：调用 `map` 方法时，调用者能够选择“算法”对每个元素进行转换。

- *模板方法*（template method）

  以 final 方法的形式定义某一算法的实现框架，该方法会调用其他的一些函数，而为了能够对这些方法的行为进行定制，子类可以对这些被调用方法进行覆写。正如 11.1 节中讲述的那样，覆写具体方法的做法既破坏了既有规则，也不安全。模板方法则更讲原则且更为安全，也正因为如此，模板方法是我最喜欢的模式之一。值得注意的是，除了定义用于覆写的抽象方法之外，我们还可以将模板方法定义成高阶函数，将具体实现传入到高阶函数中，以实现自定义功能。

- *访问者模式*（visitor）

  使用访问者模式时，我们会向某个实例中插入一个协议。这样一来，其他代码便能通过该协议访问原本不被该类型支持的内部操作。但是，该模式其实是一个很糟糕的模式，它侵犯了 public 接口，并使实现变得复杂。不过幸运的是，我们有更好的解决选项。类型定义者可以通过定义 `unapply` 或 `unapplySeq` 方法来定义一种低开销的协议，对外只暴露应该开放的内部状态。模式匹配便使用了这一功能提取匹配值并实现新的功能。类型类无法提供内部状态的访问接口，因此无法满足一些特殊的需求，不过我们还是可以使用它们为既存类型添加新的行为。当然，访问内部状态本身就是一个很糟糕的设计。

## 23.5　契约式设计带来更好的设计

我们所使用的类型能够陈述程序所允许的状态。为了验证类型未能指定的行为，我们会采用*测试驱动开发*（TDD）或其他的测试方法。在 TDD 和函数式编程还未成为主流之前，Bertrand Meyer 便提出了*契约式设计*（design by contract，DbC），Meyer 运用 Eiffel 语言（http://archive.eiffel.com/doc/manuals/technology/contract）实现了这一方法。尽管这一方法已经不再受人青睐，不过仍然出现了一些按照“在客户和服务之前建立契约”的思路构造出来的设计。思考设计时，契约式设计是一种非常有用的隐喻。我们大多数时候都会使用 DbC 术语。

模块的“契约”应指定下面三类条件。

(1) 为了能够成功地提供服务，应该为模块输入指定哪些约束？这些约束被称为前置条件。假如服务并不是以“纯”函数的形式提供，那么这些约束也许还需考虑系统需求和一些额外的数据。前置条件会对用户的行为进行约束。

(2) 当用户输入满足前置条件后，应该设计哪些约束，对模块返回的结果进行保障？这些条件被称为后置条件，它们对服务本身进行约束。

(3) 在服务执行之前与执行之后，哪些不变量必须满足要求？

除了上述三个条件之外，契约式设计要求必须以可执行代码的方式指定这些契约式约束。这样一来，这些约束在系统运行时便能自动地强制执行。假如某一条件未通过检验，系统便会立刻停止。这迫使你必须立刻找到并修复造成这一问题的错误。（我曾经参与过这样一个项目，该项目一直成功实施 DbC 设计，直到某天团队领导认为 DbC 所导致的程序中

断带来了某些“不便利”的地方。之后的几个月里，系统日志中写满了各种不合契约的错误信息，但没有人会费神修复这些错误。）

常见的做法是只在测试时才开启条件检查，在生产环境中则关闭这些检查。这样一来，我们便能在生产环境中移除这些检查所带来的额外开销，同时也不会由于某个条件未通过而导致生产环境中系统的崩溃。值得注意的是，actor 模型信奉 let it crash 任其崩溃原则。在系统运行时，假如某个条件未通过检查，难道不应该让系统崩溃，之后再运行时启动系统恢复吗？

尽管 Scala 并没有为契约式设计提供什么显式的支持，不过 `Predef` 对象（http://www.scala-lang.org/api/current/index.html#scala.Predef$）提供了许多可用于契约式设计的方法，它们是 `assert` 方法、`assume` 方法和 `require` 方法。下面的示例演示了如何使用 `require` 和 `assert` 方法构造契约：

```
// src/main/scala/progscala2/appdesign/dbc/BankAccount.sc

case class Money(val amount: Double) {                         // ❶
  require(amount >= 0.0, s"Negative amount $amount not allowed")

  def +  (m: Money): Money = Money(amount + m.amount)
  def -  (m: Money): Money = Money(amount - m.amount)
  def >= (m: Money): Boolean = amount >= m.amount
}

case class BankAccount(balance: Money) {

  def debit(amount: Money) = {                                 // ❷
    assert(balance >= amount,
      s"Overdrafts are not permitted, balance = $balance, debit = $amount")
    new BankAccount(balance - amount)
  }

  def credit(amount: Money) = {                                // ❸
    new BankAccount(balance + amount)
  }
}
```

❶ `Money` 类对货币金额进行了封装，该类使用 `require` 语句声明了前置条件，只允许货币金额为正数。（请参考稍后关于该程序运行分析的相关讨论。）

❷ `debit` 方法不会允许余额变成负数。这是 `BankAccount` 类必须要满足的条件，也是为什么我们使用 `assert`，而不是 `require` 语句的原因。

❸ 我们希望至少在这个未使用事务的简单的示例中，不要发生任何违反契约的事情。

我们可以尝试运行下列脚本：

```
import scala.util.Try

Seq(-10, 0, 10) foreach (i => println(f"$i%3d: ${Try(Money(i))}"))

val ba1 = BankAccount(Money(10.0))
val ba2 = ba1.credit(Money(5.0))
val ba3 = ba2.debit(Money(8.5))
```

```
val ba4 = Try(ba3.debit(Money(10.0)))

println(s"""
  |Initial state: $ba1
  |After credit of $$5.0: $ba2
  |After debit of $$8.5: $ba3
  |After debit of $$10.0: $ba4""".stripMargin)
```

println 方法将产生下列输出：

```
-10: Failure(java.lang.IllegalArgumentException:
       requirement failed: Negative amount -10.0 not allowed)
  0: Success($0.0)
 10: Success($10.0)

Initial state: BankAccount($10.0)
After credit of $5.0: BankAccount($15.0)
After debit of $8.5: BankAccount($6.5)
After debit of $10.0: Failure(java.lang.AssertionError:
  assertion failed: Overdrafts are not permitted, balance = $6.5, debit = $10.0)
```

assrt、assume 语句以及 require 方法均提供了两个可重载的实现版本。下面我们列举了 assert 的两个可重载方法：

```
final def assert(assertion: Boolean): Unit
final def assert(assertion: Boolean, message: => Any): Unit
```

假如断言参数值为 false，那么在 assert 的第二类实现方法中传入的 message 参数将出现在错误信息中。否则将使用默认消息。

assert 和 assume 方法表现相同。尽管它们的名字表明各自的目的不同，但是假如编译时使用了选项 -Xelide-below ASSERTION（此处的 ASSERTION 可以替换成某一更高的值），两者都会抛出 AssertionError 代码且两种方法调用都不会出现在字节码中。

require 方法用于对方法参数（包括了构造方法）进行测试。当测试失败时，require 方法会抛出 IllegalArgumentException 异常，而相关的代码并不会受到 -Xelide-below 编译选项的影响。因此，在我们编写的 Money 类型中，require 检查永远无法关闭。即使是在关闭了 assert 和 assume 检测的生产环境中，require 检查也不会关闭。假如你并不希望使用这种无法关闭的检测，请选择 assert 或 assume 方法。

最理想的情况是能够通过类型系统对所有条件进行检测。但 Scala 类型系统目前尚无法做到这点。因此，我们仍应使用 TDD（或者 TDD 的变种）和契约式设计所带来的断言检查构造正确的软件。

## 23.6 帕特农神庙架构

通用语言（ubiquitous language）是面向对象设计中最具诱惑力的思想，它为我们描绘了这样一幅美景：所有的团队成员，无论你是商业利益相关人士还是 QA，为了能够更有效地进行沟通，都使用相同的领域语言（2003 年，Eric Evans 在其编写的《领域驱动设计》一书中提出了“通用语言”这一名词）。而实践环节中，这意味着所有的领域概念都将被实

现成具有行为的各个类型，代码中也将大量使用到这些类型。

函数式代码不同于这类代码，在函数式代码中你只能看到极少数的“原子”数据类型和容器，所有这些类型和容器都具备精准的代数属性。函数式代码精简而又准确，利用这些优势，我们的代码既能按期完成也能满足代码质量的要求。

在实现一些真实世界的领域概念时，由于这些概念与当前的情景有天然的相关性，我们因此会碰到一些难题。对于 `Taxpayer`，你的想法与我的想法不同，这是因为我们需要实现不同的使用场景（也可以称作“用户描述、需求”等）。如果我们深入探讨这些问题的本质，我们会发现我们需要从数据存储中读取一些数值，并根据税法中的一些特定规则对这些数值执行数学运算，之后再提供计算结果报表。所有这些程序都是 CRUD 程序（CRUD 是 create、read、update 和 delete 的简写，分别代表创建、读取、更新和删除操作）。你也许以为我是在夸夸其谈，但事实上我只是稍微夸大了一点点。

根据下列规则来决定是否要在代码中实现某一领域概念。

- 与元组或 map 这样的通用类型进行比较：
  - 使用该领域概念能显著提高代码的封装性。
  - 使用该领域概念能够清楚地阐述代码的作用。
- 这一概念具有定义良好的数学属性。
- 这一概念能够提高代码的正确性。例如：与那些更为通用的类型相比，新的概念能够限定允许值。

我们是否应该为金钱定义一个专门的类型呢？由于金钱具有一些良好的可定义属性，因此我们应该为其定义类型。通过 `Money` 类型，我们可以执行运算，并制定需要遵循的规则，如封装的 `Double` 或 `BigDecimal` 值是非负数，根据标准的会计条例舍入运算执行到“分”这一货币单位等等。

`USZipCode` 具有定义良好的属性。尽管不会对邮编执行任何运算，但是我们可以将邮编的允许值限定为 US 邮局服务所能识别的五个字符或五个字符外加四个数字。

为了提高编码效率，如果条件允许，我会使用 value 类（`AnyVal` 类型的子类）来实现这些类型。

不过，在实现 `Taxpayer` 和其他一些模糊概念时，我会使用由健值对组成的 Map 对象、集合或仅仅包含了待实现用例中所需数据字段信息的元组。

不过在从这些通用语言中获益的同时，会不会有什么弊端呢？对此，我花费了一些时间进行思考，考虑只汲取了通用语言益处的架构样式。



下文将要探讨的内容只是某一想法的粗略梗概，该想法几乎是纯理论的且没有验证。

架构包含四层内容。

- 通用语言的 DSL 层

  该层被用于阐述用例。由于 UI（用户界面）也是一类交流工具，因此同样被作为一门语言被包含在这层。

- DSL 库

  DSL 的实现层，包含一些领域概念的实现类型、UI 实现等内容。

- 用例逻辑

  用户逻辑层包含用于实现各个用例的函数式代码。这些代码主要依赖于标准库类型、领域相关类型，而代码本身也会尽可能的集中、简洁。由于这些代码非常简洁，大多数用例的实现代码就是系统的一个单独切面。

- 核心库

  这一层包含 Scala 标准库、Akka 库、Play 库、日志 API、数据库访问相关库等核心库，以及从用例实现中提取出来的可重用代码。

下面这张图让我联想起希腊神庙的经典形象，其中每个用例都构成了一列柱子。正因为如此，请允许我自负的将该架构称为帕特农架构（请参考图 23-1）。

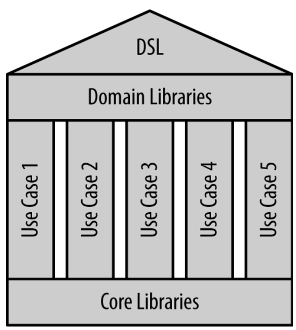



图 23-1：帕特农神庙架构

在帕特农架构中，神庙的根基代表核心库，列柱代表用例代码。柱上楣构代表支持领域的库，包括 DSL 实现和 UI。顶部的三角楣饰代表 DSL 代码，该代码由用户编写，用于实现每个用例。了解更多寺庙术语，请参见维基百科（http://en.wikipedia.org/wiki/Ancient_Greek_temple）。

由于用例代码列柱看上去排斥重用，因此这种想法虽然很新奇，但也带来了一些争议。架构顶部提供了可重用的领域相关代码库，而底部同样提供了很多可重用的库。这使得该架构看上去像“烟囱”反模式（https://en.wikipedia.org/wiki/Stovepipe_system）。

不过，我们作出的每个设计选择都同时具有优势和劣势。重用带来的好处是能够消除冗余，劣势则是容易造成代码拥堵点，也就是说，许多代码路径会同时经过相同的可重用对象，这在面向对象的系统中尤为突出。假如这些对象中包含可变状态，便会导致一些问题。形成代码拥堵点之后，用例之间的逻辑的剥离会变得困难。在这种情况下，我们也很难进行独立开发；由于这些用例会横跨多个进程，这也限制了代码水平扩展的能力。

同样的，就像本书中出现的一些示例一样，每个用例所对应的函数式代码应该非常短小，因此，我们无需花费精力移除那些普通的复制。相反的，这些简单的、适当的数据流逻辑易于理解、测试和升级。

在 20.3 节中我们曾经讲解了一个使用外部 DSL 的工资系统，我们将重新引用该示例。这个示例略微有些复杂，我们会读取一个记录了一组用户信息的数据文件，文件中的数据以逗号分隔，之后我们会根据文件内容构造字符串，并将这些字符串放入 DSL 之中进行分析并生成我们需要的数据结构，最后我们将实现两个用例：报告每个员工工资的检查单和报告工资发放期的总体情况。在实际的应用程序中应用这些中间字符串并没有什么意义，不过在这个示例中，这样做能够让我们在不做任何修改的情况下便能复用之前的 DSL，而且这些中间字符串有助于解释程序的要点：

```
// src/main/scala/progscala2/appdesign/parthenon/PayrollUseCases.scala
package progscala2.appdesign.parthenon
import progscala2.dsls.payroll.parsercomb.dsl.PayrollParser
import progscala2.dsls.payroll.common._

object PayrollParthenon {                                          // ❶
  val dsl = """biweekly {
      federal tax          %f  percent,
      state tax            %f  percent,
      insurance premiums   %f  dollars,
      retirement savings   %f  percent
    }"""
                                                                   // ❷
  private def readData(inputFileName: String): Seq[(String, Money, String)] =
    for {
      line <- scala.io.Source.fromFile(inputFileName).getLines.toVector
      if line.matches("\\s*#.*") == false  // skip comments
    } yield toRule(line)

  private def toRule(line: String): (String, Money, String) = {      // ❸
    val Array(name, salary, fedTax, stateTax, insurance, retirement) =
      line.split("""\s*,\s*""")
    val ruleString = dsl.format(
      fedTax.toDouble, stateTax.toDouble,
      insurance.toDouble, retirement.toDouble)
    (name, Money(salary.toDouble), ruleString)
  }

  private val parser = new PayrollParser                           // ❹

  private def toDeduction(rule: String) =
    parser.parseAll(parser.biweekly, rule).get

  private type EmployeeData = (String, Money, Deductions)           // ❺
                                                                   // ❻
  private def processRules(inputFileName: String): Seq[EmployeeData] = {
    val data = readData(inputFileName)
    for {
      (name, salary, rule) <- data
      deductions      = toDeduction(rule)
    } yield (name, salary, toDeduction(rule))
```

```
  }
                                                              // ❼
  def biweeklyPayrollPerEmployeeReportUseCase(data: Seq[EmployeeData]): Unit ={
    val fmt  = "%-10s %6.2f  %5.2f  %5.2f\n"
    val head = "%-10s %-7s  %-5s    %s\n"
    println("\nBiweekly Payroll:")
    printf(head, "Name", "Gross", "Net", "Deductions")
    printf(head, "----", "-----", "---", "----------")
    for {
      (name, salary, deductions) <- data
      gross = deductions.gross(salary.amount)
      net   = deductions.net(salary.amount)
    } printf(fmt, name, gross, net, gross - net)
  }
                                                              // ❽
  def biweeklyPayrollTotalsReportUseCase(data: Seq[EmployeeData]): Unit = {
    val (gross, net) = (data foldLeft (0.0, 0.0)) {
      case ((gross, net), (name, salary, deductions)) =>
      val g = deductions.gross(salary.amount)
      val n = deductions.net(salary.amount)
      (gross + g, net + n)
    }
    printf("\nBiweekly Totals: Gross %7.2f, Net %6.2f, Deductions: %6.2f\n",
      gross, net, gross - net)
  }

  def main(args: Array[String]) = {
    val inputFileName =
      if (args.length > 0) args(0) else "misc/parthenon-payroll.txt"
    val data = processRules(inputFileName)

    biweeklyPayrollTotalsReportUseCase(data)
    biweeklyPayrollPerEmployeeReportUseCase(data)
  }
}
```

❶ 首先我们使用 DSL 语言定义了一个格式化字符串，字符串中的实际数值会在运行时填充进去。

❷ 从输入文件中读取数据，移除文件中注释行内容（注释行起始位置会出现任意长度的空白字符，之后紧跟着 # 字符），之后使用 DSL 将每条员工记录转换成一个规则。考虑到示例的简单性，我们忽略了错误处理，还复用了契约式设计相关章节中使用的 `Money` 类（我们没有重复列出 `Money` 类的实现代码）。

❸ 将每条记录分解成一个个字段，并将数字都转化为 `Double` 类型，之后我们使用规则字符串为每位员工生成相关信息，最后返回员工姓名、工资和规则。

❹ 像往常一样，我们构造了一个 DSL 解释器，并使用该解释器对规则字符串进行解析。

❺ 为了提高代码的可读性，我们定义了一个类型别名。这是一个比较合算的做法，因为我们只需要在内部使用该别名。

❻ 读取数据文件，并从中抽取出每位员工的名字、工资以及扣除金额信息。

❼ 用例：汇报每位员工在每两周一次的发薪日所获取的工资、净收入以及工资扣项。

❽ 用例：汇报每两周一次的发薪日中，所有员工获得的工资总额、净收入总额以及总共的工资扣项。

默认情况下，程序会加载 misc 文件内的一个数据文件。假如你使用命令 `run-main progscala2.appdesign.parthenon.PayrollParthenon` 在 `sbt` 中运行该程序，这将得到 `main` 方法执行的两个用例的输出信息，输出如下所示：

```
Biweekly Totals: Gross 19230.77, Net 12723.08, Deductions: 6507.69

Biweekly Payroll:
Name       Gross    Net      Deductions
----       -----    ---      ----------
Joe CEO    7692.31  5184.62  2507.69
Jane CFO   6923.08  4457.69  2465.38
Phil Coder 4615.38  3080.77  1534.62
```

尽管该程序还有大量的改良空间，不过由于该程序仅仅用来说明我们可以使用简短的、彼此无关的代码“柱子”实现真实的用例，因此我们不对其进行修改。这些代码从“顶层”库中选取了一些领域概念，并从“底层”库提供的 Scala API 中挑选出一些核心基础设施。

## 23.7 本章回顾与下一章提要

我们讨论了应用设计过程中出现的许多范式问题，其中包括设计模式和契约式设计。我们也深入讲解了我个人一直思考的架构模型，该模型被称为帕特农神庙架构。

本书的最后一章会介绍 Scala 提供的反射和元编程机制。

第 24 章

# 元编程：宏与反射

*元编程*（metaprogramming）是一种操纵程序而不是操纵数据的编程。在一些语言中，编程和元编程之间的差异并不是那么显著。例如：Lisp 的方言使用相同的 S 表达式来表示代码和数据，这种性质称为*同像性*（homoiconicity）。操作代码非常简单，也很常见。但在 Java 和 Scala 这样的静态类型语言中，元编程则是不常见的。但实际上元编程对解决许多设计问题非常有用。

“反射”一词有时也用于指元编程。对于 Scala 的反射库来说的确如此。但是，有时反射只表示狭义上的代码的运行“内省”之意。

在 Scala 这样的语言中，代码先被编译，然后才能运行。但许多动态类型语言则直接解释执行，两者对应的编译时元编程与运行时元编程有明显的区别。在编译时元编程中，所有的调用均发生在编译前或编译过程中。经典的 C 语言预处理就是一个例子，它在编译前对源代码进行转换。

Scala 的元编程可以发生在编译时，使用宏来支持。宏有点像受约束的编译器插件，因为它们操纵从所解析的源代码中生成的抽象语法树（AST）。宏在生产字节码的最后编译阶段之前调用，以操纵 AST。

Java 反射库和 Scala 的扩展库提供运行时反射。

Scala 的反射 API（包括宏），是 Scala 发展最快的一个部分。由于其目标变换很快，我们将重点放在最稳定的部分，即运行时反射和宏工具 quasiquote。不过，在结束时，我们会给出一个完整的使用当前宏 API 的示例。

新一代的宏功能正处于开发阶段。该项目被称为 Scala Meta（http://scalamacros.org）。在写作本书的时候，Scala Meta 的预览版即将发布。你应该找找关于它的最新资料，因为它们会出现在后续版本的 Scala 中。对于当前的 Scala 2.10 和 2.11 版本的宏的实现，请访问

http://scalamacros.org 和 Macro Paradise（http://docs.scala-lang.org/overviews/macros/paradise.html），后者是当前宏系统的孵化项目。

首先，我们将介绍一些有用的 REPL 工具，来了解表达式的类型。然后我们会探索运行时反射，其次是 quasiquote，最后讨论宏的示例。

## 24.1 用于理解类型的工具

REPL 有个 : type 命令，用来打印类型信息：

```
scala> if (true) false else 11.1
res0: AnyVal = false

scala> :type if (true) false else 11.1
AnyVal

scala> :type -v if (true) false else 11.1
// Type signature
AnyVal

// Internal Type structure
TypeRef(TypeSymbol(abstract class AnyVal extends Any))
```

`:type` 命令只显示类型。通常情况下，REPL 会显示类型信息，不过，`-v`（verbose，表示详细信息）选项也显示"类型内部结构"。`scala.reflect.api.Types.TypeRef`（http://www.scala-lang.org/api/current/scala-reflect/#scala.reflect.api.Types$TypeRef）和 `scala.reflect.api.Symbols.TypeSymbol`（http://www.scala-lang.org/api/current/scala-reflect/#scala.reflect.api.Symbols$TypeSymbol）这几个类型定义在反射 API 中。现在反射 API 已经从核心标准库中独立了出来。Scaladoc 可以在 http://www.scala-lang.org/api/current/scala-reflect/#package 找到。

## 24.2 运行时反射

编译时反射用于操纵代码，而运行时反射则主要用来"调整"语言的语义（在一定范围内），或着加载编译时未知的代码，即所谓的*极端滞后绑定*（extreme late binding）。

例如，属性或命令行参数用于动态地指定实例的种类以实现特定的功能。反射 API 用于从 `CLASSPATH` 能找到的字节码中，定位相应的类型；如果可以找到对应的类型，还要构造相应的实例。IDE 等工具可以使用反射来发现和加载插件。IDE 还经常使用反射了解有关项目和库的代码，以支持代码补全和类型检查等功能。另外字节码工具也可以使用反射来寻找安全漏洞等问题。

### 24.2.1 类型反射

你可以在 `java.lang.Class`（http://docs.oracle.com/javase/8/docs/api/java/lang/Class.html）的方法中使用 Java 的反射 API。

```
// src/main/scala/progscala2/metaprogramming/reflect.sc

scala> import scala.language.existentials
import scala.language.existentials

scala> trait T[A] {
     |   val vT: A
     |   def mT = vT
     | }
defined trait T

scala> class C(foo: Int) extends T[String] {
     |   val vT = "T"
     |   val vC = "C"
     |   def mC = vC
     |
     |   class C2
     | }
defined class C

scala> val c = new C(3)
c: C = $anon$1@5a58e6a4

scala> val clazz = classOf[C]              // Scala 方法: classOf[C]
clazz: Class[C] = class C

scala> val clazz2 = c.getClass             // java.lang.Object中的方法
clazz2: Class[_ <: C] = class $anon$1

scala> val name  = clazz.getName
name: String = C

scala> val methods = clazz.getMethods
methods: Array[java.lang.reflect.Method] =
  Array(public java.lang.String C.mC(), public java.lang.Object C.vT(), ...)

scala> val ctors = clazz.getConstructors
ctors: Array[java.lang.reflect.Constructor[_]] = Array(public C(int))

scala> val fields = clazz.getFields
fields: Array[java.lang.reflect.Field] = Array()

scala> val annos = clazz.getAnnotations
annos: Array[java.lang.annotation.Annotation] = Array()

scala> val parentInterfaces = clazz.getInterfaces
parentInterfaces: Array[Class[_]] = Array(interface T)

scala> val superClass = clazz.getSuperclass
superClass: Class[_ >: C] = class java.lang.Object

scala> val typeParams = clazz.getTypeParameters
typeParams: Array[java.lang.reflect.TypeVariable[Class[C]]] = Array()
```

这些方法只能用于 AnyRef 的子类型。需要注意的是，getFields 似乎不能识别类型 C 中

Scala 类型字段！

Predef 定义了一些方法，用来测试对象是否属于某个类，或者将对象转为某个类型：

```
scala> c.isInstanceOf[String]
<console>:13: warning: fruitless type test: a value of type C cannot
  also be a String (the underlying of String)
              c.isInstanceOf[String]
                             ^

res0: Boolean = false

scala> c.isInstanceOf[C]
res1: Boolean = true

scala> c.asInstanceOf[T[AnyRef]]
res2: T[AnyRef] = C@499a497b
```

完成这些任务的 Java 操作符是关键字。Scala 的方法名故意起得十分冗长，目的是抑制它们的使用！ Scala 的其他特性，尤其是模式匹配，是完成这一目的更好的选择。

### 24.2.2 ClassTag、TypeTag与Manifest

Scala 2.11 核心库有一个小的反射 API，而功能更先进的反射特性则在独立的库中。下面我们来探讨核心库中的 ClassTag（http://www.scala-lang.org/api/current/#scala.reflect.ClassTag），这是一个用来保留被类型擦除去掉信息的工具。类型擦除是 JVM 的一个“特性”，在实例化参数类型的实例时，不保留类型参数的值。ClassTag 的一个重要用途是构造出正确的 AnyRef 子类型的 Java 数组。以下是一个改编自 ClassTag 的 Scaladoc（http://www.scala-lang.org/api/current/#scala.reflect.ClassTag）帮助页面的例子：

```
// src/main/scala/progscala2/metaprogramming/mkArray.sc
scala> import scala.reflect.ClassTag
import scala.reflect.ClassTag

scala> def mkArray[T : ClassTag](elems: T*) = Array[T](elems: _*)
mkArray: [T](elems: T*)(implicit evidence$1: scala.reflect.ClassTag[T])Array[T]

scala> mkArray(1, 2, 3)
res0: Array[Int] = Array(1, 2, 3)

scala> mkArray("one", "two", "three")
res1: Array[String] = Array(one, two, three)

scala> mkArray(1, "two", 3.14)
<console>:10: warning: a type was inferred to be `Any`;
  this may indicate a programming error.
              mkArray(1, "two", 3.14)
                      ^
res2: Array[Any] = Array(1, two, 3.14)
```

这里对 AnyRef 使用 Array.apply 方法（http://www.scala-lang.org/api/current/#scala.Array$），该方法还有一个参数列表，其中包含一个隐含的 ClassTag 参数。 编译器会利用它知道的

类型信息来构造隐含的 ClassTag。然而，对于过去就已经构造完成的列表，其关键的类型信息已经丢失。如果你传入了集合，然后居于栈中很深的位置的某些方法希望用 ClassTag 来自省时，就会出现这种问题。构造集合与相应的 ClassTag 必须在同一作用域，然后将它们一起传递给方法。ClassTag 通过隐含参数进行传递。我们稍后会再回来讨论这个问题。

因此，ClassTags 不能从字节码中将类型信息“复活”，但它们可以在类型信息被擦除之前捕获和利用这些类型信息。

ClassTag 实际上是一个较弱的版本的 scala.reflect.api.TypeTags#TypeTag（http://www.scala-lang.org/api/current/scala-reflect/#scala.reflect.api.TypeTags$TypeTag）。后者是一个独立的 API，它保留了完整的编译时间信息（不久我们就将利用它），而 ClassTag 只返回了运行时的信息。此外，还有一个 scala.reflect.api.TypeTags#WeakTypeTag（http://www.scala-lang.org/api/current/scala-reflect/#scala.reflect.api.TypeTags$WeakTypeTag）用于抽象类型。Scala doc（http://docs.scala-lang.org/overviews/reflection/typetags-manifests.html）中对其有详细的说明。

需要注意的是，在 Scala 2.10 引入 TypeTag 和 ClassTag 之前，reflect 包中还有一些老的类型用于相同的用途，称为 Manifest。这些类型已经废弃，你会在旧的源代码中见到。但是，你应该使用较新的特性。

## 24.3　Scala的高级运行时反射API

反射 API 的其余部分支持更丰富的运行时反射和编译时的宏。它包括用于表示抽象语法树和其他上下文的类型。这部分 API 作为一个单独的 JAR 文件分发，是我们在构建 sbt 时包含的依赖之一。这个 API 的全部细节都在 Scaladoc（http://docs.scala-lang.org/overviews/reflection/overview.html）中有对应的描述。我们将讨论这个非常大的 API 的核心理念，并列举几个类型运行时内省的例子：

```
// src/main/scala/progscala2/metaprogramming/match-type-tags.sc

import scala.reflect.runtime.universe._                           // ❶

def toType2[T](t: T)(implicit tag: TypeTag[T]): Type = tag.tpe    // ❷
def toType[T : TypeTag](t: T): Type = typeOf[T]                   // ❸
```

❶ 导入 runtime.universe 中的定义，其类型是 scala.reflect.api.JavaUniverse（http://www.scala-lang.org/api/current/scala-reflect/index.html#scala.reflect.api.JavaUniverse）。它为目标平台暴露了反射语言元素和一些便利的方法。

❷ 使用了 TypeTag[T]（http://www.scala-lang.org/api/current/scala-reflect/index.html#scala.reflect.api.TypeTags$TypeTag）类型的隐含参数，然后返回该参数的类型。

❸ 绑定上下文，这是一种更简便的方法。typeOf[T] 方法是 implicitly[TypeTag[T]].tpe 的简写。

回想一下，TypeTag 保留了完整的编译时类型信息，同时 ClassTag 只保留了运行时类型信息。

我们用几个类型来测试一下这些方法：

```
scala> toType(1)
res1: reflect.runtime.universe.Type = Int

scala> toType(true)
res2: reflect.runtime.universe.Type = Boolean

scala> toType(Seq(1, true, 3.14))
<console>:12: warning: a type was inferred to be `AnyVal`;
  this may indicate a programming error.
              toType(Seq(1, true, 3.14))
                         ^
res3: reflect.runtime.universe.Type = Seq[AnyVal]

scala> toType((i: Int) => i.toString)
res4: reflect.runtime.universe.Type = Int => java.lang.String
```

注意到，这里得到了参数化类型的类型参数。这修复了我们在使用 ClassTag 时的 bug。我们会从现在开始忽略 AnyVal 警告。

我们可以比较类型是否相等或是否具有父子关系：

```
toType(1) =:= typeOf[AnyVal]                    // false
toType(1) =:= toType(1)                         // true
toType(1) =:= toType(true)                      // false

toType(1) <:< typeOf[AnyVal]                    // true
toType(1) <:< toType(1)                         // true
toType(1) <:< toType(true)                      // false

typeOf[Seq[Int]] =:= typeOf[Seq[Any]]           // false
typeOf[Seq[Int]] <:< typeOf[Seq[Any]]           // true
```

我们一直通过调用 tpe 方法来从 TypeTag 中获取 Type（http://www.scala-lang.org/api/current/scala-reflect/index.html#scala.reflect.api.Types$Type）。你也可以用 typeTag（http://www.scala-lang.org/api/current/scala-reflect/#scala.reflect.api.TypeTags$TypeTag）辅助函数得到类型的层次：

```
typeTag[Int]          // reflect.runtime.universe.TypeTag[Int] = TypeTag[Int]
typeTag[Seq[Int]]     // ...TypeTag[Seq[Int]] = TypeTag[scala.Seq[Int]]
```

在 10.1.1 节中，我们讨论了函数的协变和逆变。现在我们用新工具来获取这些详细信息：

```
// src/main/scala/progscala2/metaprogramming/func.sc

class CSuper                { def msuper() = println("CSuper") }
class C      extends CSuper { def m()      = println("C") }
class CSub   extends C      { def msub()   = println("CSub") }

typeOf[C      => C     ] =:= typeOf[C => C]      // true   ❶
typeOf[CSuper => CSub  ] =:= typeOf[C => C]      // false
typeOf[CSub   => CSuper] =:= typeOf[C => C]      // false
```

```
typeOf[C      => C     ] <:< typeOf[C => C]      // true   ❷
typeOf[CSuper => CSub  ] <:< typeOf[C => C]      // true   ❸
typeOf[CSub   => CSuper] <:< typeOf[C => C]      // false  ❹
```

❶ 除了严格匹配以外，其他均不相等。

❷ 任何类型都是其自身的子类型，所以这一条成立。

❸ 参数是 C 的父类型，满足参数的逆变性。返回值是 C 的子类型，满足返回值的协变性。因此这一条成立。

❹ 同时违反了关于函数参数和返回值的规则。

> 当一个类型是另一个类型的子类型时，忘记相应的规则？使用 typeOf 或 toType 方法和 <:< 来弄明白吧。

现在，考虑一下，我们可以从类型中获取哪些信息。首先，Type 返回了 TypeRef（http://www.scala-lang.org/api/current/scala-reflect/index.html#scala.reflect.api.Types$TypeRef）的实例，所以我们用提取器来确定其“前缀”，即类型的符号（名称），和它需要的类型参数：

```
def toTypeRefInfo[T : TypeTag](x: T): (Type, Symbol, Seq[Type]) = {
  val TypeRef(pre, typName, parems) = toType(x)
  (pre, typName, parems)
}
```

元组中的 Type 和 Symbol 类型均定义在 reflect.runtime.universe 中，但不要与 scala.Symbol 混淆。

```
toTypeRefInfo(1)                        // (scala.type, class Int, List())
toTypeRefInfo(true)                     // (scala.type, class Boolean, List())
toTypeRefInfo(Seq(1, true, 3.14))       // (scala.collection.type, trait Seq,
                                        //    List(AnyVal))
toTypeRefInfo((i: Int) => i.toString)   // (scala.type, trait Function1,
                                        //    List(Int, java.lang.String))
```

注意 Seq 的“前缀”scala.collection.type 与其他例子的“前缀”scala.type 的不同。正如我们预想的那样，Seq 和 Function1 都具有非空的类型参数列表。

我们甚至可以通过 TypeApi（http://www.scala-lang.org/api/current/scala-reflect/index.html#scala.reflect.api.Types$TypeApi）得到更多信息。我们在 REPL 中尝试使用 Seq，查看返回的类型。这里我们会省略过长的输出：

```
scala> val ts = toType(Seq(1, true, 3.14))
ts: reflect.runtime.universe.Type = Seq[AnyVal]

scala> ts.typeSymbol
res0: reflect.runtime.universe.Symbol = trait Seq

scala> ts.erasure
res1: reflect.runtime.universe.Type = Seq[Any]
```

```
scala> ts.typeArgs
res2: List[reflect.runtime.universe.Type] = List(AnyVal)

scala> ts.baseClasses
res4: List[reflect.runtime.universe.Symbol] =
  List(trait Seq, trait SeqLike, trait GenSeq, trait GenSeqLike, ...)

scala> ts.companion
res5: reflect.runtime.universe.Type = scala.collection.Seq.type

scala> ts.decls
res6: reflect.runtime.universe.MemberScope = SynchronizedOps(
  method $init$, method companion, method seq)

scala> ts.members
res7: reflect.runtime.universe.MemberScope = Scopes(
  method seq, method companion, method $init$, method toString, ...)
```

上述代码中大部分是自解释的。`companion` 方法返回了伴随类型的类型值，`decls` 返回了其自身在 `Seq` 中的定义，而成员返回了被继承的所有声明。

了解更多示例，请参见概述（http://docs.scala-lang.org/overviews/reflection/overview.html）和反射的 Scaladoc（http://www.scala-lang.org/api/current/scala-reflect/#package）。

## 24.4 宏

Scala 当前的宏已经在许多先进的工具包中为设计问题提供了巧妙的解决方案。但是，使用它时，必须要了解编译器的内部结构，如编译器使用的抽象语法树（AST）。因此，Scala Meta 项目（http://scalameta.org）旨在实现一个新的宏系统，可以避免与编译细节的耦合，以降低用户的学习负担。它也将从第一个宏系统的设计中总结经验教训。

由于 Scala Meta 目前尚不可用，而当前的系统最终会废弃，因此我们不会讨论细节，但我们将在结束时讨论一个示例。有一个特性有望保持在 Scala Meta 中保持相对不变，即 quasiquote。quasiquote 使用内插字符串的方式，使得操纵 AST 变得容易得多。quasiquote 消除了使用旧 API 编写宏时需要的大量样板代码和细节知识。在 Scaladoc（http://docs.scala-lang.org/overviews/quasiquotes/intro.html）中有 quasiquote 的文档。

需要注意的是，尽管与 2.10 版本相比，Scala 2.11 版本的 API 只有一点变化，但是本章剩下的例子只在 Scala 2.11 中工作。具体而言，我们很快就会看到一个新的辅助方法 `showCode`。我们在后面关于宏的示例中还会提到 API 上的改变。

我们试试其中的一些特性：

```
// src/main/scala/progscala2/metaprogramming/quasiquotes.sc

import reflect.runtime.universe._                              // ❶

import reflect.runtime.currentMirror                           // ❷
import tools.reflect.ToolBox
val toolbox = currentMirror.mkToolBox()
```

❶ 导入 quasiquote 需要的 `universe`。

❷ 引入了方便使用的 toolbox。

根据你要构建的 AST 树的种类，存在几种方法可以构造 quasiquote。我们为表达式使用一般的形式 `q"…"` 和 `tq"…"`。其中完整的选项列表和示例可以在 Scala 文档（http://docs.scala-lang.org/overviews/quasiquotes/syntax-summary.html）中查找。

```
scala> val C = q"case class C(s: String)"
C: reflect.runtime.universe.ClassDef =
case class C extends scala.Product with scala.Serializable {
  <caseaccessor> <paramaccessor> val s: String = _;
  def <init>(s: String) = {
    super.<init>();
    ()
  }
}

scala> showCode(C)
res0: String = case class C(s: String)

scala> showRaw(C)
res1: String = ClassDef(Modifiers(CASE), TypeName("C"), List(), ...)
```

`showCode` 方法打印出的字符串与 Scala 声明的语法类似（在这个例子中，与 Scala 语法一模一样），`showRaw` 则根据 AST 树打印出类型。

`q` 被用来表示一般的 quasiquote，`tq` 则用来构造类型树，如：

```
scala> val  q =  q"List[String]"
q: reflect.runtime.universe.Tree = List[String]

scala> val tq = tq"List[String]"
tq: reflect.runtime.universe.Tree = List[String]

scala> showRaw(q)
res2: String = TypeApply(Ident(TermName("List")),
  List(Ident(TypeName("String"))))

scala> showRaw(tq)
res2: String = AppliedTypeTree(Ident(TypeName("List")),
  List(Ident(TypeName("String"))))

scala> q equalsStructure tq
res4: Boolean = false
```

用 `showRaw` 可看到它们实际上是不同的。`scala.reflect.api.Trees#TypeApplyExtractor`（http://www.scala-lang.org/api/current/scala-reflect/index.html#scala.reflect.api.Trees$TypeApplyExtractor）的 Scaladoc 页面解释了这种差异。`TypeApply`（http://www.scala-lang.org/api/current/scala-reflect/index.html#scala.reflect.api.Trees$TypeApply）对应术语中指定的类型，如 `def foo[T](t: T) = ...` 中的 `foo[T]`，而 `AppliedTypeTree`（http://www.scala-lang.org/api/current/scala-reflect/index.html#scala.reflect.api.Trees$AppliedTypeTree）则用于类型声明，如 `val t: T` 中的 `T`。

用 equalsStructure 测试是否相等。

你可以使用字符串插值 ${...}，即所谓的 unquoting，将其他的 quasiquote 扩展为一个 quasiquote：

```
scala> Seq(tq"Int", tq"String") map { param =>
     |    q"case class C(s: $param)"
     |  } foreach { q =>
     |    println(showCode(q))
     |  }
case class C(s: Int)
case class C(s: String)
```

因此，我们可以对代码生成做参数化！请注意我们使用的是类型 quasiquote（tq"..."），这是因为 param 函数的参数是用来做类型声明的。试着用 showRaw 替换 showCode，然后用 q 或字符串（如 "Int"）来替换 tq，再比较这两个替换的输出。

有时，做内插时正常的值会被“提升”为 quasiquote：

```
scala> val list = Seq(1,2,3,4)
scala> val fmt = "%d, %d, %d, %d"
scala> val printq = q"println($fmt, ..$list)"
```

.. $list 形式的语法将列表扩展为由逗号分隔的值。（还有一个 ...$list 用于处理序列的序列。）在这里，我们用它来调用可变参数函数 println。相反的过程是“降低”，通常在模式匹配中用于 quasiquote 的字符串。

```
scala> val q"${i: Int} + ${d: Double}" = q"1 + 3.14"
i: Int = 1
d: Double = 3.14
```

还有一些其他类型的 quasiquote 的：cq 为 case 语句生成树，fq 为 for 表达式生成树，pq 为模式匹配表达式生成树。了解详细示例，请参见 Scaladoc（http://docs.scala-lang.org/overviews/quasiquotes/syntax-summary.html）。

### 24.4.1 宏的示例：强制不变性

当我们在 23.5 节讨论了契约式设计，其中提到了契约的一个性质——不变性：在每一个方法调用前后以及阶段改变前后，不变性都应该保持。下面我们用宏来实现强制不变性。

回想一下，宏是一个受限的编译器插件，在编译过程的中间阶段调用。这就要求宏必须与使用宏的代码提前编译且必须分开进行编译。我们将在一个源文件中实现宏，并在 ScalaTest 测试文件中使用它，以满足这一要求。这种方法是可行的，因为 sbt 将测试代码和主代码分开编译。

此外，宏的实现遵循一定的约定，我们很快就会看到。以下是宏 invariant 的源代码：

```
// src/main/scala/progscala2/metaprogramming/invariant.scala
package metaprogramming
import reflect.runtime.universe._                              // ❶
import scala.language.experimental.macros
```

```
import scala.reflect.macros.blackbox.Context                  // ❷

/**
 * 使用宏语法和quasiquotes编写宏时
 * 要求谓词的测试结果在评估每个语句之前必须为真。
  */
object invariant {                                            // ❸
  case class InvariantFailure(msg: String) extends RuntimeException(msg)

  def apply[T](predicate: => Boolean)(block: => T): T = macro impl   // ❹

  def impl(c: Context)(predicate: c.Tree)(block: c.Tree) = {    // ❺
    import c.universe._                                       // ❻
    val predStr = showCode(predicate)                         // ❼
    val q"..$stmts" = block                                   // ❽
    val invariantStmts = stmts.flatMap { stmt =>              // ❾
      val msg = s"FAILURE! $predStr == false, for statement: " + showCode(stmt)
      val tif = q"throw new metaprogramming.invariant.InvariantFailure($msg)"
      val predq2 = q"if (false == $predicate) $tif"
      List(q"{ val tmp = $stmt; $predq2; tmp };")
    }
    val tif = q"throw new metaprogramming.invariant.InvariantFailure($predStr)"
    val predq = q"if (false == $predicate) $tif"
    q"$predq; ..$invariantStmts"                              // ❿
  }
}
```

❶ 导入所需的反射和宏。我们正在构建一个“黑盒”的宏；它不会改变其包围的表达式的签名。（详细信息见文档（http://docs.scala-lang.org/overviews/macros/blackbox-whitebox.html）。）

❷ 对于 Scala 2.10 版，用 `scala.reflect.macros.Context` 代替。

❸ `invariant.apply` 方法用来包装我们想要强制规定为不可变的表达式。如果失败，将会抛出 `InvariantFailure` 异常。

❹ 宏始终以共有方法为开始，该方法由客户端代码调用且方法体包含 `macro impl`。在这种情况下，要传入两个参数：一个谓词，用来对第二个参数中的每个语句进行测试，另一个是用来执行的代码块。

❺ 方法 `impl` 带的参数与 `apply` 的参数对应，其中每一个都是从表达式中生成的抽象语法树。类型 `c.Tree` 是 `Context`（http://www.scala-lang.org/api/current/scala-reflect/index.html#scala.reflect.macros.blackbox.Context）对象中的一个路径依赖类型，是 `impl` 的第一个参数。

❻ 我们需要使用与上下文对应的 universe，所以需要导入 universe 的成员。

❼ 为 `predicate` 创建错误信息的字符串。

❽ 用模式匹配将 `block` 转为语句序列。

❾ 对语句做扁平映射（flat map），对每一个语句做修改，以捕捉它们的返回值。然后检查谓词（如果失败，抛出 `InvariantFailure` 异常），然后返回结果。

❿ 重新连接各个语句，在前面加上对谓词的简单判断，并返回修改后的 AST。

如果没有 quasiquote，这份实现会难写得多，因为我们必须知道 AST 具体实现的细节，以及如何操纵 AST 树。

下面我们看一个例子，并在以下 ScalaTest 中进行测试。在这里我们将使用 `Variable` 类，该类中有 2 个可变字段，之后我们将强制字符串字段 `s` 成为不可变的字段：

```
// src/test/scala/progscala2/metaprogramming/InvariantSpec.scala
package metaprogramming
import reflect.runtime.universe._
import org.scalatest.FunSpec

class InvariantSpec extends FunSpec {
  case class Variable(var i: Int, var s: String)

  describe ("invariant.apply") {
    def succeed() = {                                          // ❶
      val v = Variable(0, "Hello!")
      val i1 = invariant(v.s == "Hello!") {                    // ❷
        v.i += 1
        v.i += 1
        v.i
      }
      assert (i1 === 2)
    }

    it ("should not fail if the invariant holds") { succeed() }

    it ("should return the value returned by the expressions") { succeed() }

    it ("should fail if the invariant is broken") {
      intercept[invariant.InvariantFailure] {                  // ❸
        val v = Variable(0, "Hello!")
        invariant(v.s == "Hello!") {
          v.i += 1
          v.s = "Goodbye!"
          v.i += 1
        }
      }
    }
  }
}
```

❶ 辅助方法，用来检查 invariant 包含的值。

❷ 在代码块中执行语句时，需要保持字符串字段不变。

❸ 预期会抛出 `InvariantFailure`，因为字符串被修改了。

可以在注释中去掉 `intercept[…]` 这一行和相应的大括号。之后测试将会失败，并出现以下错误信息：

```
[info] - should fail if the invariant is broken *** FAILED ***
[info]   metaprogramming.invariant$InvariantFailure:
  FAILURE! v.s.==("Hello!") == false, for statement: v.`s_=`("Goodbye!")
```

对于失败的谓词和触发该失败的语句，我们可以显示可读性很强的消息，这一点非常强大。整个实现只需要几十行代码，却演示了宏的强大功能。

如果手写以上的所有代码，你会发现 InvariantSpec 的第一个测试看起来很像下面的代码。在下面这段代码中，我将循环中的代码提取为一个辅助方法 fail，用来避免冗余：

```
def fail[T](predStr: String, stmtStr: String): Nothing = {
  val msg = s"FAILURE! $predStr == false, for statement: $stmtStr"
  throw new metaprogramming.invariant.InvariantFailure(msg)
}
val v = Variable(0, "Hello!")
val i1 = {
  if (v.s != "Hello!") fail("v.s == \"Hello!\"", "")
  val tmp1 = v.i += 1
  if (v.s != "Hello!") fail("v.s == \"Hello!\"", "v.i += 1")
  val tmp2 = v.i += 1
  if (v.s != "Hello!") fail("v.s == \"Hello!\"", "v.i += 1")
  val tmp3 = v.i
  if (v.s != "Hello!") fail("v.s == \"Hello!\"", "v.i")
  tmp3
}
```

在 quasiquote 的文档中存在其他一些很有用的示例，例如：在代码块中，每个语句执行之前打印调试语句。

### 24.4.2 关于宏的最后思考

宏的强大功能相当诱人，但开发、调试和维护宏也很具挑战性。你可以在第三方库中找到很多使用宏的例子。不过还是要记住，所有反射 API，特别是有关宏的包，都被认为是实验性的，因而它们需要继续快速地发展。

## 24.5 本章回顾与下一章提要

如果已经读到这里，你现在已经掌握了 Scala 语言的所有主要特性，并且也掌握了如何最好地运用它们。我希望这里的代码示例可以作为你自己的项目模板。如果需要更多关于不同类型的应用程序和工具集的示例，请参见 Typesafe 网站（http://typesafe.com/activator）上的 Activator 项目。Typesafe 还为开发者提供了关于 Scala、Akka、Play 和其他工具的订阅资源，这样的工具变得越来越多。Typesafe 也提供培训和咨询。

在接下来的几年，Scala 将会如何改变？自《Scala 程序设计（第 1 版）》出版以来，无论是语言的成熟度还是行业的应用程度上，Scala 都发生了巨大的变化。预计 Scala 的应用会继续增长，尤其在大数据背景下的今天。Scala 本身的演化已经稳定。即使是宏，也将在一年或两年内稳定下来。Scala 及相关的外围生态系统的许多工作将定位于提高性能、减少 bug、废弃语言上的“瘤”，以及提高 Scala 外围工具上，如 IDE 对 Scala 的支持等。

Martin Odersky 正在开发一门类似 Scala 的语言，该语言基于一个新的类型系统，称为 DOT，意为依赖对象类型（dependent object typing）。这有可能成为 Scala 3.0（更多信息，请参见 DOT 幻灯片（http://lampwww.epfl.ch/~amin/dot/fool_slides.pdf）和 PDF（http://

www.cs.uwm.edu/~boyland/fool2012/papers/fool2012_submission_3.pdf)）。

DOT 基于依赖类型（dependent typing），这是目前类型理论中最先进的，允许“包含三个元素的数组”这样的概念作为一个类型。目前，大多数语言的类型系统还不能将尺寸约束作为一种类型的一部分。为什么这个非常重要？它可以推动我们向可证明的正确的程序迈进一步，在这样的程序中，类型相当于定理，而程序是定理的证明（见维基百科（http://en.wikipedia.org/wiki/Curry%E2%80%93Howard_correspondence)）。

这门新的语言也将在其他方面简化类型系统，并去掉语言上的“瘤”。但这至少需要几年时间。

在此期间，你可以使用 Scala 来改善你创建应用程序的方式，同时利用成熟的 Java 生态系统的丰富性，或者更进一步讲，你可以通过 Scala 的 scala.js（http://www.scala-js.org）利用充满活力的 JavaScript 生态系统。我希望《Scala 程序设计（第 2 版）》能够帮助读者们获得成功。

# 附录A

# 参考文献

Abelson, Harold, Gerald Jay Sussman, and Julie Sussman, *Structure and Interpretation of Computer Programs*. The MIT Press, 1996.

Agha, Gul, *Actors*. The MIT Press, 1987.

"Akka: Build powerful concurrent & distributed applications more easily," http://akka.io.Alexander, Alvin, *Scala Cookbook: Recipes for Object-Oriented and Functional Programming*. O'Reilly Media, 2013.

Algebird (https://github.com/twitter/algebird)

Allen, Jamie, *Effective Akka*. O'Reilly Media, 2013.

Antlr (http://www.antlr.org/)

Barr, Michael and Charles Wells, "Category Theory for Computing Science" (http://www.tac.mta.ca/tac/reprints/articles/22/tr22.pdf), 1998.

Behavior-Driven Development (http://behaviour-driven.org/)

Bloch, Joshua, *Effective Java (Second Edition)*. Addison-Wesley, 2008.

Bird, Richard, *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010.

Bjarnason, Rúnar Óli, "Stackless Scala and Free Monads" (http://blog.higher-order.com/assets/trampolines.pdf).

Bonér, Jonas, "Real-World Scala: Dependency Injection (DI)" (http://jonasboner.com/2008/10/06/real-world-scala-dependency-injection-di/).

Bruce, Kim, Martin Odersky, and Philip Wadler, "A Statically Safe Alternative to Virtual Types,"

*Proc. ECOOP'98*, E. Jul (Ed.), LNCS 1445, pp. 523–549, Springer-Verlag, 1998.

“Building bug-free O-O software: An introduction to Design by Contract” (http://archive.eiffel.com/doc/manuals/technology/contract/).

Chiusano, Paul and Rúnar Bjarnason, Functional Programming in Scala. Manning Publications, 2013.

Dean, Jeffrey and Sanjay Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters” (https://www.usenix.org/legacy/event/osdi04/tech/full_papers/dean/dean.pdf).

Dzilums, Lauris, “Awesome Scala” (https://github.com/lauris/awesome-scala).

Easterbrook, Steve, “An introduction to Category Theory for Software Engineers” (http://www.cs.toronto.edu/~sme/presentations/cat101.pdf).

Eiffel Software(http://eiffel.com)

Effective Scala(http://twitter.github.io/effectivescala/)

Evans, Eric, *Domain Driven Design*. Prentice-Hall, 2003.

Extension Methods (C# Programming Guide) (http://msdn.microsoft.com/en-us/library/bb383977.aspx)

Finagle (https://twitter.github.io/finagle/)

Ford, Bryan, “The Packrat Parsing and Parsing Expression Grammars Page” (http://pdos.csail.mit.edu/~baford/packrat/).

Fowler, Martin, *Domain-Specific Languages*. Addison-Wesley, 2010.

Ghosh, Debasish, *DSLs in Action*. Manning Press, 2010.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides ( “Gang of Four” ), *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Gradle (http://www.gradle.org/)

Guice (http://code.google.com/p/google-guice/)

Hadoop (http://hadoop.apache.org)

Haller, Philipp and Martin Odersky, “Actors That Unify Threads and Events” (http://lamp.epfl.ch/~phaller/doc/haller07coord.pdf).

Hewitt, Carl, Peter Bishop, and Richard Steiger, “A Universal Modular Actor Formalism for Artificial Intelligence,” *IJCAI'73*, August 20-23, 1973, Stanford, California, USA.

Hewitt, Carl, “Actor Model of Computation” (http://arxiv.org/pdf/1008.1459.pdf), 2014.

Hoare, Tony, “Null References: The Billion Dollar Mistake,” http://bit.ly/null-refs-th.

Hofer, Christian, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors, “Polymorphic

Embedding of DSLs" (http://www.daimi.au.dk/~ko/papers/gpce50_hofer.pdf), GPCE'08, October 19–23, 2008, Nashville, Tennessee.

Hypertext Transfer Protocol — HTTP/1.1 (http://www.w3.org/Protocols/rfc2616/rfc2616.html).

Hunt, Andrew and Dave Thomas, *The Pragmatic Programmer*. Addison-Wesley, 2000.

Introducing Spring Scala (http://spring.io/blog/2012/12/10/introducing-spring-scala)

Iry, James, "Phantom Types in Haskell and Scala" (http://james-iry.blogspot.ch/2010/10/phantom-types-in-haskell-and-scala.html).

Java Platform SE 8 API (http://docs.oracle.com/javase/8/docs/api/)

The Java Tutorials. Lesson: Java Regular Expressions (http://docs.oracle.com/javase/tutorial/essential/regex/).

Laddad, Ramnivas, *AspectJ in Action (Second Edition)*. Manning Press, 2009.

Lawvere, F. William and Stephen H. Schanuel, *Conceptual Mathematics, A First Introduction to Categories*. Cambridge University Press, 2009.

Lipovaca, Miran, *Learn You a Haskell for Great Good*! No Starch Press, 2011.

Liskov Substitution Principle (http://en.wikipedia.org/wiki/Liskov_substitution_principle).

Malawski, Konrad, "Scala's Types of Types" (http://ktoso.github.io/scala-types-of-types/).

Marick, Brian,*Functional Programming for the Object-Oriented Programmer*. Leanpub, 2012.

Martin, Robert C., *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.

Meyer, Bertrand,*Object-Oriented Software Construction (Second Edition)*. Prentice Hall, 1997.

Naftalin, Maurice and Philip Wadler, *Java Generics and Collections*. O'Reilly Media, 2006.

Nilsson, Rickard, *ScalaCheck: The Definitive Guide*. Artima Press, 2013.

Odersky, Martin and Matthias Zenger, "Scalable Component Abstractions," *OOP-SLA*'05, October 16–20, 2005, San Diego, California, USA.

Odersky, Martin, Lex Spoon, and Bill Venners, "How to Write an Equality Method in Java" (http://www.artima.com/lejava/articles/equality.html).

Odersky, Martin, Lex Spoon, and Bill Venners, *Programming in Scala, Second Edition*. Artima Press, 2010.

Okasaki, Chris, *Purely Functional Data Structures*. Cambridge University Press, 1998.

O'Sullivan, Bryan, John Goerzen, and Don Steward, *Real World Haskell*. O'Reilly Media, 2009.

Parsing Expression Grammar (http://en.wikipedia.org/wiki/Parsing_expression_grammar)

Paul, Thomas, "Working with Money in Java" (http://www.javaranch.com/journal/2003/07/

MoneyInJava.html).

Phillips, Andrew and Nermin Serifovic, *Scala Puzzlers*. Artima Press, 2014.

Pierce, Benjamin C., *Types and Programming Languages*. The MIT Press, 2002.

Rabhi, Fethi and Guy Lapalme, *Algorithms: A Functional Programming Approach*. Addison-Wesley, 1999.

Roestenburg, Raymond, Rob Bakker, and Rob Williams, *Akka in Action*. Manning, 2014.

S-99: Ninety-Nine Scala Problems (http://aperiodic.net/phil/scala/s-99/)

Sargent, Will, "Error Handling in Scala" (http://tersesystems.com/2012/12/27/error-handling-in-scala/)

Scala Automatic Resource Management (http://jsuereth.com/scala-arm/)

ScalaCheck (http://scalacheck.org/)

The Scala Language Specification (http://www.scala-lang.org/docu/files/ScalaReference.pdf)

The Scala Library (http://www.scala-lang.org/api/current/)

The Scala Programming Language (http://www.scala-lang.org/)

ScalaTest (http://www.scalatest.org/)

Scalaz (https://github.com/scalaz/scalaz)

Scalding (https://github.com/twitter/scalding)

Shapeless: Generic Programming for Scala (https://github.com/milessabin/shapeless)

Simple Build Tool (http://www.scala-sbt.org)

SIP-15: Value Classes (http://docs.scala-lang.org/sips/pending/value-classes.html)

Spark (http://spark.apache.org)

Specs2 (http://etorreborre.github.io/specs2/)

Spiewak, Daniel, "What is Hindley-Milner? (and why is it cool?)" (http://www.codecommit.com/blog/scala/what-is-hindley-milner-and-why-is-it-cool).

Spiewak, Daniel, "Interop Between Java and Scala" (http://www.codecommit.com/blog/java/interop-between-java-and-scala).

Spiewak, Daniel, "The Magic Behind Parser Combinators" (http://www.codecommit.com/blog/scala/the-magic-behind-parser-combinators).

The Spring Framework (http://spring.io)

Suereth, Joshua, *Scala in Depth*. Manning Press, 2012.

Suereth, Joshua and Matthew Farwell, *SBT in Action*. Manning Press, 2013.

Szyperski, Clemens, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Limited, 1998.

Taylor, Chris, “The Algebra of Algebraic Data Types” (http://chris-taylor.github.io/blog/2013/02/10/the-algebra-of-algebraic-data-types/).

Turbak, Franklyn, David Gifford, and Mark A. Sheldon, *Design Concepts of Programming Languages*. The MIT Press, 2008.

Typesafe, Inc. (http://typesafe.com)

Van Roy, Peter and Seif Haridi, *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004.

Wadler, Philip, “The Expression Problem” (http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt).

Walters, R.F.C., *Categories and Computer Science*. Cambridge University Press, 1992.

Wampler, Dean, *Introduction to Functional Programming for Java Developers*. O’Reilly Media, 2011.

Theoretical Computer Science: “What’s new in purely functional data structures since Okasaki?” (http://cstheory.stackexchange.com/questions/1539/whats-new-in-purely-functional-data-structures-since-okasaki).

White, Tom, Hadoop: *The Definitive Guide, Third Edition*. O’Reilly Media, 2012.

Wirfs-Brock, Rebecca and Alan McKean, *Object Design: Roles, Responsibilities, and Collaborations*. Pearson Education, 2003.

Wyatt, Derek, *Akka Concurrency*. Artima Press, 2013.

# 作者简介

Dean Wampler 博士是 Typesafe 公司的大数据产品架构师。他大力倡导使用 Scala 作为大数据应用软件的最佳开发工具。Dean 曾与他人合编了《Hive 编程指南》一书，也是 O'Reilly 出版的《面向 Java 开发者的函数式编程》的作者。Dean 为若干开源项目贡献了代码，也是多个技术大会以及芝加哥用户组的组织者。在 Twitter 上可以搜索 @deanwampler 找到他。

Alex Payne 是一名程序员、作家，同时也是一位天使投资人，主要投资初创公司。在担任在线银行服务 Simple 的 CTO 和在 Twitter 担任平台负责人时，他部署了 Scala 开发的应用程序。Alex 是年度新兴语言大会的组织者，该大会是新编程语言和开发工具的展示平台。他本人也是技术与商业大会的演讲者。你可以搜索 @al3x 在 Twitter 上找到他，或者访问他的个人网站 https://al3x.net。

# 封面介绍

本书封面上的动物是马来貘（貘紫檀），也称作亚洲貘。这是一种体色黑白的蹄类哺乳动物，有着猪一样的滚圆、敦实的体型。马来貘体长约 6~8 英尺，体重 550~700 磅，是四种貘类中体型最大的，生活在东南亚的热带雨林地区。

马来貘的体型十分惊人：它的前半身与后腿是纯黑色的，而白色的腹部看上去像是一个马鞍。在月光照耀下的密林中，这样的身体特征有利于马来貘进行完美的伪装。它的皮很厚，尾部粗短，鼻口部短小却非常灵活。尽管看上去非常笨重，但马来貘攀爬和跑动起来都非常灵敏。

马来貘是独居动物，主要在夜间活动。它的视力一般很差，所以依靠嗅觉和听觉在区域里寻找食物或者追踪其他的马来貘，用高亢的口哨声来交流。马来貘的捕食者包括虎、豹以及人类。由于栖息地的破坏和人类过度地捕猎，马来貘已经被列为濒危物种。

封面图片来自多弗尔画报档案。

# Scala程序设计（第2版）

Scala具备现代对象模型、函数式编程以及先进类型系统的所有优势，是一门可以满足现代软件工程师需求的语言。本书通过大量的代码示例，向读者全面展示了在Scala语言生态环境下如何高效地编写代码，同时阐明了Scala是目前编写高扩展性和以数据为中心的应用软件的最佳语言。

在第1版的基础之上，第2版介绍了Scala的最新语言特性，新添了模式匹配、推导式以及高级函数式编程等知识。通过本书，读者还能学会如何使用Scala命令行工具、第三方工具、库以及适用于编辑器和IDE的Scala相关插件。本书既适合Scala初学者入门，也适合经验丰富的Scala开发者参考。

**通过阅读本书，你可以：**

- **利用Scala简洁灵活的语法，提高编程效率；**
- **深入学习函数式编程的基本技能和高级技能；**
- **使用Scala函数式组合器，构造“杀手级”大数据应用；**
- **使用Scala提供的trait类型实现mixin组合，使用模式匹配实现数据抽取功能；**
- **学习Scala语言中复杂的类型系统，了解函数式编程和面向对象编程中的概念；**
- **深入学习包括Akka的Scala并发工具；**
- **掌握如何开发丰富的领域特定语言。**

作为一本强调数据科学的图书，本书中出现的代码示例均保存在公开的Github仓库中。通过立即可启动的虚拟机，这些示例代码可以很容易地获得。该虚拟机中预装了一组IPython Notebook，为我们提供方便的交互式学习环境。

**Dean Wampler**博士，Typesafe公司的大数据架构师。Typesafe使用Scala、函数式编程、Spark、Hadoop以及Akka技术编写数据处理与分析的工具和服务。Dean是《面向Java开发者的函数式编程》的作者，同时也与他人合著了《Hive编程指南》一书。

**Alex Payne**是Twitter的平台组长。在Alex开发的服务基础上，其他的程序开发者构造了一套备受欢迎的社交消息服务。此前，Alex还为政治竞选、公益性组织以及初创企业编写过一些Web应用。

SCALA/JAVA/PROGRAMMING LANGUAGES

**封面设计**：Karen Montgomery　张健

图灵社区：iTuring.cn
热线：(010)51095186转600

| 分类建议 | 计算机 / 程序设计 / Scala |
|---|---|

**人民邮电出版社网址**：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版



ISBN 978-7-115-41681-0
ISBN 978-7-115-41681-0
定价：109.00元